Pedro Nuno Pereira Severino

Licenciado em Engenharia Informática

Aceleração da Classificação de Documentos com Recurso a GPU

Dissertação para obtenção do Grau de Mestre em Engenharia Informática

Orientador: Prof. Doutor Pedro Abílio Duarte de Medeiros

Júri:

Presidente: Prof. Doutor Carlos Damásio

Arguentes: Prof. Doutor Salvador Abreu

Vogais: Prof. Doutor Pedro Medeiros

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Setembro, 2012

Aceleração da Classificação de Documentos com Recurso a GPU

Gostava de dedicar esta dissertação aos meus pais por me terem ensinado que existe sempre uma recompensa para o nosso trabalho árduo e que podemos ter tudo o que desejamos desde que estejamos dispostos a lutar por isso. Também gostava de dedicar a tese a toda a minha família e amigos que todos os dias me mostram que existe muito mais para além do trabalho árduo e também devemos desfrutar a nossa vida ao passar bons momentos com os mesmos.

Agradecimentos

À Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa, especificamente aos professores que me forneceram, durante todo o curso, todos os conhecimentos necessários para conceber este Projecto; Ao Professor Doutor Pedro Medeiros, por toda a sua ajuda e disponibilidade para me orientar no decorrer desta dissertação.

Aos meus colegas de trabalho João Moura e João Nogueira, por toda a ajuda durante o período de adaptação a um ambiente de trabalho empresarial, ajuda a enquadrar-me com o projecto desenvolvido antes da execução desta dissertação e ajuda na elaboração da mesma.

Ao meus vários colegas de curso, mais especificamente ao Ricardo Marques, pela sua disponibilidade para longas discussões sobre a elaboração da dissertação, durante o decorrer da mesma; À minha família e amigos, por toda a ajuda, compreensão e apoio em todo o curso.

Os meus mais sinceros agradecimentos!

Resumo

O objectivo desta dissertação é explorar formas de diminuir o tempo gasto na extracção de características relevantes(*features*) de textos disponíveis na Internet.

Esta diminuição é obtida paralelizando parte dos algoritmos usados; as versões paralelizadas são vocacionadas para a execução em GPUs. São concebidas, implementadas e testadas versões paralelas da fase da determinação da relevância de *features* num sujeito/objecto a ser analisado. Nesta parte do trabalho a principal contribuição é uma implementação do algoritmo de extracção de *features* adaptado a GPUs com ênfase nas rotinas de manipulação de grafos. Em objectos mais complexos a implementação atingiu tempos de execução 16 vezes inferiores à implementação sequencial em CPU.

Foi também desenvolvido um servidor concorrente que reside no GPU e que oferece um conjunto de serviços relacionados com o processamento das *features* mais relevantes. Esse servidor faz uma gestão integrada dos recursos de computação existentes no GPU. A avaliação deste servidor foi feita sujeitando-o a diferentes misturas de sujeitos e com um ritmo de chegada de pedidos crescente. Considerando o critério mais relevante o número de pedidos cujo processamento excede um tempo limite, a solução baseada no servidor GPU começa a exceder esse tempo quando o ritmo de chegada de sujeitos ultrapassa os 70 pedidos/s, enquanto que na versão multi CPU isso acontece quando o ritmo atinge perto de 30 pedidos/s. Neste âmbito foi desenvolvida uma infra-estrutura que pode ser reaproveitada sempre que seja conveniente usar o GPU no paradigma do processamento de pedidos em lote.

Esta tese foi desenvolvida num contexto empresarial, e do trabalho realizado resultam contributos para o melhoramento dos produtos da empresa, bem como da viabilidade técnica e económica do uso de GPUs em diversos contextos relevantes para a empresa. Neste contexto a versão definitiva da tese omite alguma informação.

Palavras-chave: OpenCL; Computação Paralela; GPU; Classificação de Documentos

Abstract

The main goal of this dissertation is to design and implement solutions that reduce the execution time of algorithms for feature extraction in texts available in the Internet.

This time reduction will be achieved by the parallelization of parts of the algorithms; the target of the parallelized versions will be GPUs. In this work, we designed, implemented and accessed parallel versions of the algorithms used for finding the relevance of features. In this part, the main contribution is an implementation of the feature extraction algorithm targeted to GPUs, more specifically the routines used for the manipulation of the graph used to represent a subject's structure. In complex Web documents the GPU version achieved execution times 16x smaller than the ones of the CPU sequential version.

Another contribution was a concurrent server that resides in the GPU and offers services related with the processing of relevant features. The server makes an integrated management of the GPU resources. The assessment of the server was made by stressing it with a growing rate of requests; the criterion used for comparison was the number of requests dropped because a threshold time of execution was exceeded. The GPU server begins to drop requests when the request rate is 70 req/s; the multi-CPU version drops requests when the rate is 30 req/s. The framework used for building the GPU version can be used in other contexts where the paradigm of submitting a bulk of requests to the GPU can be applied.

The work conducting to this dissertation was made within an enterprise context; the work done made sound contributions for the enhancement of the company's products and also assessed the technical and economical viability of the use of GPUs in several contexts relevant to the company. Due to the context of the dissertation elaboration, some information is not present in this final version of the document.

Keywords: OpenCL; Parallel Computing; GPU; Web page classification

Conteúdo

Lista de Figuras

1

Enquadramento

O presente trabalho é realizado no âmbito da Unidade Curricular Dissertação, do 2º ano de Mestrado em Engenharia Informática da Faculdade de Ciências e Tecnologia (FCT), da Universidade Nova de Lisboa (UNL) com a finalidade da obtenção do grau de Mestre em Engenharia Informática.

A dissertação realizada enquadra-se num projecto empresarial que utiliza algoritmos de extracção de características relevantes (daqui para a frente designadas por *features*)) de documentos.

A área de trabalho é a re-implementação de algoritmos de extracção de *features*, criando um esquema de paralelização sobre GPU. O projecto tem como principal objectivo a aceleração do atendimento de pedidos por parte de um servidor com recurso a GPU. Espera-se que esta aceleração se traduza tanto a nível de tempos de execução reduzidos como a nível do *throughput*[1] por parte de um servidor concorrente.

Este capítulo começa por fazer uma descrição detalhada do problema para o qual está a ser proposta uma nova solução. Logo após a descrição do problema é feita uma apresentação detalhada da solução actualmente em uso, que actualmente apenas se baseia no uso de CPU. Por último, são referidas as contribuições que derivam do desenvolvimento deste projecto e uma pequena secção onde se faz uma enumeração dos vários tópicos que vão ser abordados ao longo da dissertação, para além da organização dos mesmo.

1.1 Classificação

A extracção de *features* é uma acção de extrema importância, sendo necessária em vários cenários. Este tipo de algoritmo é utilizado em vários projectos de *data mining*.

[1] Número de pedidos executado por segundo

Apesar de todo este foco no produto da empresa, os resultados desenvolvidos, também podem ser utilizados fora deste âmbito para outro tipo de projectos, podendo estes ser empresariais ou académicos.

As *features* para o caso específico da empresa traduzem-se em sequências de palavras, embora o projecto possa abranger outro tipo de *features*, visto que o algoritmo de extracção é genérico, carecendo apenas de acertos específicos conforme o tipo de *feature* em análise. Estas sequências de uma ou mais palavras são chamadas n-gramas.

O primeiro passo é a identificação do conjunto de n-gramas mais significativos de um documento, sendo estes as *features*. Após extracção das *features* mais relevantes do documento são utilizadas técnicas/ferramentas de *data mining* para lidar com os dados extraídos. Apesar deste tipo de técnicas/ferramentas ser de grande interesse, esta encontra-se fora do âmbito desta dissertação e ao longo da mesma apenas serão feitas pequenas referências a este tipo de ferramenta.

1.2 Fluxo de Execução do Processo Original

Este capítulo encontra-se omitido devido a questões de confidencialidade empresarial.

1.3 Abordagem

Quando se trata de paralelizar uma aplicação, com o intuito de reduzir o tempo de execução da mesma há várias abordagens possíveis, tanto a nível de hardware como a nível de software. De seguida são apresentadas brevemente as opções feitas neste projecto. Depois fazemos uma pequena análise em relação à implementação corrente do projecto. Por último, são apresentadas os traços gerais da solução desenvolvida.

1.3.1 Hardware

Do ponto de visto do hardware, escolheu-se utilizar Graphics Processing Units(GPU), porque este tipo de dispositivo hardware tem uma melhor relação custo/desempenho em relação a outros dispositivos analisados ao longo desta dissertação. Desta escolha deriva um desafio não trivial, que é a reformulação de algoritmos para este tipo de hardware. A mudança de código que vai ser exigida torna-se interessante visto que este hardware utiliza um paradigma de paralelismo completamente diferente do habitual com o qual os programadores são confrontados.

A mudança de paradigma para esta arquitectura[ZFM] vem acompanhada de novos problemas que não seriam sequer passíveis de acontecer em CPU. Alguns dos principais focos de atenção quando se trabalha com este tipo de hardware estão na mitigação no uso de algumas operações que não estão optimizadas neste tipo de arquitecturas, como a primitiva *if-then-else*. Este tipo de hardware costuma trabalhar com o auxílio de CPU,

apesar de ter um espaço de memória completamente disjunto do mesmo, obrigando a transferências de dados que podem ser bastante demoradas.

É importante referir que este tipo de hardware tem normalmente associado custos financeiros menores do que outro tipo de processadores, tais como CPUs.

1.3.2 Software

O ambiente software escolhido para o desenvolvimento do projecto foi o OpenCL[M+09] com *bindings* para C++, visto que esta é a linguagem utilizada maioritariamente no projecto da empresa. A utilização deste modelo deve-se à vontade de fazer várias experiências do projecto sobre vários tipos de hardware, onde se destacam GPU, AMD e nVidia, auxiliados por CPU *multi-core*.

Esta plataforma exige um controlo explícito sobre o trabalho feito em GPU e também das próprias transferências de memória que acontecem entre CPU e GPU. No entanto, devido ao controlo minucioso que a plataforma obriga a ter espera-se conseguir extrair melhorias valiosas em termos de desempenho sobre o hardware que esteja subjacente à plataforma.

1.3.3 Análise da Aplicação

A aplicação que estava implementada foi sobrecarregada com uma bateria de testes para conseguir-se analisar o seu comportamento em relação aos mesmos. Na sequência dos resultados destes testes, descobriu-se que o *bottleneck*[1] da aplicação existente está localizado numa zona onde são utilizadas estruturas de dados que representam grafos. Este poderia ser definido como o principal foco do projecto, mas tendo em conta o contexto do projecto tivemos de ser mais abrangentes. Nesta fase a criação do grafo foi funcionalmente dividida em várias funções e foram definidas as estruturas de dados mais direccionadas a este problema, tendo em conta as principais características do algoritmo que estão assentes na criação do grafo.

Há o objectivo de modificar a aplicação de forma a permitir o processamento concorrente de pedidos de classificação de documentos distintas. A aplicação também tinha o objectivo de se conseguir estabelecer num servidor de maneira a atender vários pedidos de classificação. Correntemente este problema é tratado da forma mais simples possível, alocando um núcleo do CPU a cada pedido de classificação que ia chegando, mantendo os seguintes em fila de espera até haver recursos para que estes fossem atendidos.

1.3.4 Solução

O processamento de um único pedido teve como principal foco a criação do grafo, no entanto acabou por ser alargado de maneira a aumentar o uso de GPU e a diminuir a quantidade de dados transferidos para CPU aumentando a performance final do algoritmo implementado.

[1] Zona do programa onde é gasto a maior parte do tempo de execução

Apesar disto a aceleração obtida na fase final do projecto torna este *overhead*[1] aceitável, tendo em vista o resultado final. Este factor torna-se aceitável porque o hardware permite a execução de várias *threads* em simultâneo e assim diluir este *overhead* no tempo do processamento de um pedido de classificação.

Sabendo que o classificador tem como objectivo final a integração num servidor para atender vários pedidos de classificação e que o módulo deste modelo que nos preocupa é o que fica no lado do servidor, também foi desenvolvido como fase final do projecto um classificador capaz de aumentar o *throughput* sem prejudicar em demasia o tempo médio de atendimento de um pedido. Espera-se um ligeiro abrandamento de uma classificação, visto a adaptação do classificador para o atendimento de pedidos em *bulk/batch*[2].

Este tipo de técnica foi implementada simulando um servidor concorrente a partir do GPU, tendo pequenos pontos de entrada e mantendo a maior parte dos seus pormenores escondidos às *threads* clientes. Esta simulação foi adaptada de maneira algo rígida às tarefas executadas pelo classificador, no entanto futuramente espera-se desenvolver esta técnica de maneira a simular um modelo *client-server* sobre GPU para tarefas genéricas definidas pelo programador que utilize este modelo.

1.4 Contribuições

As contribuições deste projecto são: uma diminuição significativa do tempo de processamento de um documento; um aumento expressivo do número de pedidos atendidos por unidade de tempo num servidor.

Naturalmente, um ganho de desempenho desta ordem representará um trunfo comercial importante para a empresa onde o projecto foi desenvolvido, que começa a entrar numa área um pouco deslocada do seu nicho de negócio. Este trunfo deverá ter as suas principais vantagens na melhoria de performance do serviço e num custo menor em relação a outras soluções capazes de demonstrar capacidades similares às desenvolvidas.

A solução contribui com uma abstracção capaz de representar um GPU como um servidor ao uso de vários processos em CPU, podendo o GPU tratar paralelismo funcional. Esta técnica não é muito usual, sendo a estrutura final apresentada diferente de outros projecto similares, que tratam paralelismo entre funções em GPU[GGHS09] ou paralelismos entre dados[NI09, MZZ+10], podendo estes representar dados de processos diferentes.

Por outro lado, os trabalhos conducentes à dissertação de mestrado permitiram adquirir uma experiência valiosa na área da exploração de arquitecturas com múltiplos processadores, assunto da maior relevância nos dias de hoje. Acresce que os programas a que se pretende diminuir o tempo de execução não são de paralelização trivial, podendo ser classificados como irregulares [KBI+09].

[1] Gasto de um recurso, neste caso tempo, que não contribui directamente para o objectivo pretendido
[2] Em conjunto ou em lote

1.5 Organização do Documento

O capítulo que agora termina fez o enquadramento do problema a resolver e apresentou a abordagem que vai ser usada e as contribuições que se espera fazer.

Os dois capítulos seguintes têm como objectivo englobar toda a informação referente às duas principais fases do projecto, aceleração dos tempos de execução de um pedido e aumento do *throughput* do projecto no contexto de um servidor, respectivamente.

O primeiro destes capítulos começa por descrever o problema com o qual se vai lidar, seguido da apresentação do levantamento do estado da arte em relação aos recursos utilizados na solução, possíveis alternativas e as conclusões que justificam as escolhas que foram feitas. O estado da arte foca-se principalmente no hardware e software passíveis de ser utilizados. Apresentam-se de seguida algumas das estruturas de dados para representar grafos adaptadas a GPUs. O capítulo continua a descrição da solução utilizada e apresenta os pormenores mais relevantes da implementação do mesmo, terminando com uma avaliação do trabalho desenvolvido.

Segue-se o capítulo onde um GPU é utilizado como um servidor concorrente, que tem uma estrutura similar ao capítulo que o precede. Este capítulo apresenta o trabalho relacionado com esta fase, fazendo-se uma avaliação da sua aplicabilidade ao problema em causa. O capítulo segue com uma apresentação detalhada da solução desenvolvida, mostrando quais foram as principais escolhas feitas com base no problemas que foram surgindo e apresentado uma crítica às mesmas. No final do capítulo é feita uma análise sobre a implementação feita, com base num bateria de testes propositadamente criada para estudar as principais características do trabalho desenvolvido.

Por fim, encontram-se as conclusões obtidas com esta dissertação e algumas sugestões/ideias de trabalho futuro que pode vir a ser realizado tendo este projecto e as ideias desenvolvidas com o mesmo como base.

2

Aceleração de um Pedido

Todo este capítulo se centra na aceleração de um único pedido separadamente de todos os outros. Inicialmente será apresentado e discutido o problema em si, sempre com principal foco no âmbito sobre o qual o projecto irá incidir no final. Será também feita uma referência a algumas das alternativas às plataformas *hardware* e *software* utilizadas no desenvolvimento do projecto. Também será feito o levantamento e análise de algumas estruturas de dados utilizadas para representar grafos, visto que estes são parte integrante do projecto. Após isto, será apresentada a solução concebida e os resultados obtidos a partir da mesma serão analisados e criticados, terminando o capítulo com um balanço do trabalho feito nesta fase.

Na fase inicial do projecto foram feitas medições para descobrir quais as fases mais demoradas de todo o processo. Pôde-se concluir, com base na medições apresentadas na Figura 2.1, que a fase mais custosa em termos relativos de todo o processamento era o algoritmo de extracção de *features*. Após alguma análise mais minuciosa podemos concluir que a sub-tarefa mais exigente desse algoritmo seria a criação de um grafo. Esta criação do grafo corresponde a percentagens superiores a 80% do tempo total de processamento de um pedido para alguns casos.

Sendo a criação do grafo a fase mais demorada do processo foi este o principal foco desta fase do projecto. Inicialmente apenas foi focado a migração para GPU deste processo de criação do grafo. No entanto, foi necessário estender o uso do GPU a uma área mais alargada do projecto, visto que a migração do processo referido não trouxe tantas vantagens como se pensava inicialmente, devido a grandes custos na transferência e análise do grafo criado.

Portanto estendeu-se o foco de todo o trabalho conducente a este capítulo da dissertação a todo o cálculo de relevância de *features*.

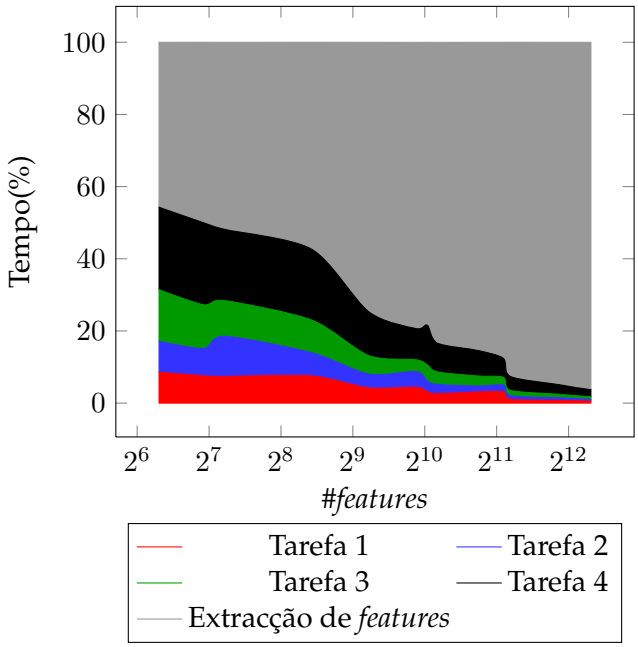

Figura 2.1: Custo Relativo das Tarefas de um Processamento

Todo o processo a ser acelerado vai desde a submissão de um conjunto de dados tratados, com uma relevância intermédia no objecto em análise até à devolução destes mesmo dados com os seus pesos/relevâncias finais no objecto, passando por isto por várias fases, de onde se destacam as seguintes:

- **Criação do grafo que define o objecto**. Vários pares de n-gramas têm os seus elementos comparados, para verificar se existe alguma relação entre eles e assim criarem algum arco entre eles no grafo que define o documento;

- **Normalização do grafo**. É feita uma análise geral aos arcos criados no grafo e com base nas ocorrências dos n-gramas relacionados pelo arco actualiza-se o peso do mesmo;

- **Algoritmo de Dispersão de Pesos**. É feita uma análise à relevância provisória dos vários n-gramas e com base no grafo anteriormente criado estes são recalculados com base num algoritmo cíclico, que é uma variação do algoritmo HITS[DHH+02];

- **Normalização de Factores**. Esta é a segunda fase do algoritmo de extracção de *features*, que tem como função fazer uma normalização dos factores de cada nó após a fase de dispersão de pesos.

A extracção de *features* superficialmente descrita no conjunto de tarefas anteriormente apresentado poderá aparecer referida como processo WHAKA, visto que este é uma nomenclatura definida dentro da empresa para este processo.

2.1 Estado da Arte

Neste sub-capítulo começa-se por discutir o hardware no qual se pretende executar o sistema de análise de páginas e justificar a escolha do mesmo. Seguidamente analisam-se os diferentes ambientes, linguagens e bibliotecas que podem ser usados para desenvolver aplicações que corram no hardware em causa. No final do capítulo faz-se uma referência às várias estruturas de dados utilizadas para representar grafos, uma vez que este tipo de estruturas é relevante para a análise do conteúdo de documentos.

2.1.1 Hardware Alvo

Apesar da exploração de paralelismo ser possível em qualquer um dos tipos de hardware apresentados, eles tendem a abordar este mesmo paralelismo de maneira diferente, podendo o mesmo ter de ser expressado de maneiras bastante diferentes para cada opção.

Existem dispositivos hardware com as mais variadas arquitecturas que tentam beneficiar com o paralelismos das aplicações. O hardware típico utilizado nesta área são CPUs *multi-core*, mas outros alvos têm sido reconhecidos pela comunidade, tais como GPU ou FPGA. Cada um dos precedentes é composto por várias unidades de processamento, o que permite a exploração do paralelismo numa vasta gama de aplicações, fazendo-as ter tempos de execução inferiores ao das versões sequenciais.

2.1.1.1 CPU *Multi-core*

Os CPUs(Central Processing Units) tem tido uma grande evolução nos últimos anos. Inicialmente a evolução deste tipo de hardware baseava-se na melhoria das velocidades de processamento do CPU, no entanto esta abordagem começou a exibir problemas. O aumento destas velocidades tornou-se muito difícil, porque este aumento implica um grande consumo de energia e dificuldades no arrefecimento do dispositivo.

Foi pensada e implementada uma nova abordagem no desenvolvimento do CPU. É aqui que entra a computação paralela como factor que influenciou a evolução deste tipo de hardware. A arquitectura que se tem vindo a desenvolver nos últimos modelos de CPU baseia-se em unidades de processamento um pouco mais lentas do que as suas antecessoras, mas em vez de uma única unidade de processamento, existem agora várias no mesmo CPU(CPU *Multi-core*). Com este novo tipo de arquitectura é mais fácil dissipar o calor proveniente dos CPU e ter melhor aproveitamento da energia gasta no processamento.

A arquitectura apresentada na Figura 2.2 é a de um CPU topo de gama nos dias de hoje. Este CPU contém 6 núcleos com relógios que vão de 3.2 a 3.47 GHz. Pode-se ver que cada *core* têm 2 conjuntos de registos(*hyper threading*[MBH+02]) e suporta operações em modo Single Instruction Multiple Data(SIMD), isto é, a mesma operação é aplicada aos 4 elementos de um vector, tendo cada elemento 32 bits.

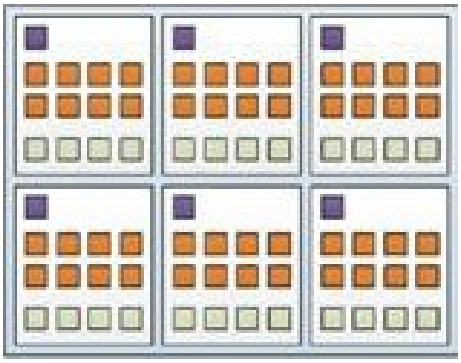

Figura 2.2: Arquitectura do Intel i7 de 6 núcleos. Retirado de [GKHM11]

Este tipo de hardware tem uma boa capacidade de resposta, mas apesar do uso intensivo de *caches* acaba por ser limitado pelo ritmo a que consegue extrair dados da memória.

Este hardware está presente em qualquer portátil, computador de secretária ou servidor nos dias de hoje, pelo que existem múltiplos tipos de abordagem para explorar o paralelismo no mesmo.

2.1.1.2 GPGPU

GPGPU(General-Purpose computation on Graphics Processing Units) é um conceito desenvolvido a partir de GPUs básicos e com base na capacidade de processamento deste tipo de dispositivos hardware. Ao longo do tempo foram desenvolvidos GPU para acelerar as computações gráficas em computadores, tirando este grande encargo da alçada dos CPU. Os GPUs podem estar localizados em vários locais numa máquina, o típico é serem encontrados em placas de vídeo, mas também podem estar implantados nas próprias motherboards da máquina ou até já em alguns CPU específicos - 2.1.1.4.

Os GPUs têm uma arquitectura muito própria. A sua arquitectura é composta por vários grupos de unidades de processamento, o que lhes dá grandes vantagens especialmente em relação a aplicações que façam processamento que se encaixe no modelo SIMD (Single Instruction Multiple Data) definido por Flynn em [Fly72]. Enquanto os CPUs utilizam técnicas como o uso de cache e hardware bastante rápido, os GPU utilizam técnicas de aceleramento que consistem no uso de programas compostos por milhares de *threads*. Uma troca de contexto praticamente sem custos, para compensar o *overhead* de leituras e escritas em memória, torna esta abordagem bastante interessante e benéfica para determinadas aplicações.

A arquitectura de um GPU da AMD, que contém 24 *cores* é apresentada na Figura 2.3. Pode-se ver que cada *core* têm vários conjuntos de registos. Esta informação permite perceber que o hardware tem um suporte bastante alargado para operações SIMD e uma grande capacidade para suporte de *threads* executados em paralelo, devido ao número de registos presentes no *chip*.

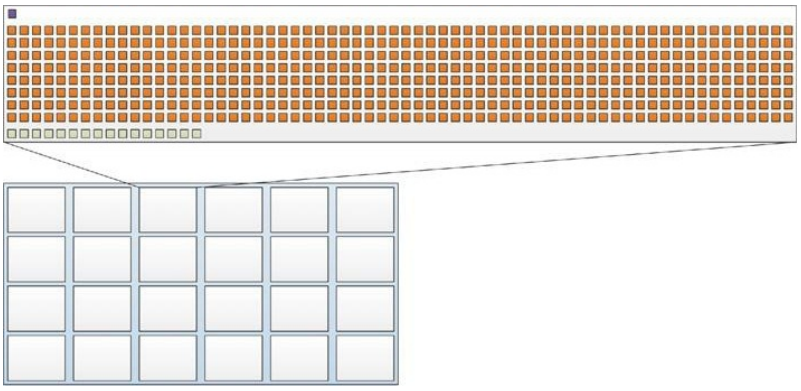

Figura 2.3: Arquitectura de uma AMD Radeon^{TM} HD6970. Retirado de [GKHM11]

Inicialmente este tipo de hardware servia para desempenhar funções como renderização e mapeamento de texturas. Posteriormente foram adicionadas mais algumas funções a este hardware, especificamente capacidades de cálculo para rotações e translações geométricas de vértices em diferentes sistemas de coordenadas. A evolução continua e cada vez mais operações são suportadas pelos GPU, onde hoje se destacam operações tais como criação dinâmica de *threads*, operação suportada pela nova arquitectura da GK110[Cor12] da nVidia.

A comunidade científica acabou por reconhecer o grande potencial deste hardware e começou a desenvolver cada vez mais aplicações de carácter geral sobre GPU. Normalmente estas aplicações trabalham com grandes volumes de dados, com computações pesadas e independentes entre os vários dados de entrada ou tarefas do problema. Alguns problemas que são obviamente beneficiados com o uso de GPU são aplicações que trabalham sobre vectores/matrizes, como uma simples soma de matrizes, um problema muitas vezes apresentado na introdução a este tipo de arquitectura.

No entanto, o trabalho com este tipo de *hardware*, independentemente da abstracção que a plataforma utilizada consegue dar, tende a obrigar o programador a ter um bom conhecimento sobre o controlo e fluxo das operações feitas no GPU. Mantendo fora do âmbito soluções mais recentes que tentam fundir CPU e GPU no mesmo *chip*, o GPU tem uma memória completamente separada e independente da memória do CPU. Este facto obriga a que a troca de informação entre os dois tipos de processadores normalmente recorra ao uso de um canal de transferência de dados que costuma trazer grandes *overheads* associados, principalmente devido à latência. Isto obriga o programador a ter sempre em conta os benefícios que podem advir do uso deste *hardware* em função dos custos que estão intrínsecos a este tipo de transferências de dados.

Um outro factor da arquitectura extremamente importante e que deve ser tido em conta são as leituras e escritas feitas por grupos de *threads* do GPU. Por norma os grupos podem ser divididos em pequenos conjuntos de *threads* que executam paralelamente

dentro do mesmo multi-processador - *warps* ou *wavefronts*, para nVidia e AMD respectivamente. Estes pequenos conjuntos executam as funções de leitura/escrita sobre a memória do GPU em conjunto, ou seja quando escrevam ou lêem numa zona de memória contígua isto traduz-se numa única operação de escrita ou leitura, isto em GPU é designado como uma operação de leitura/escrita coalescente. Se a operação for sobre n zonas de memória contíguas este traduz-se na melhor das hipóteses em n operações de leitura/escrita sobre a memória global do dispositivo, que é a que tem maiores latências no dispositivo.

Também para evitar as latências que advém do uso de memória global são utilizados outros tipos de memória com latências menores, estes são normalmente acessíveis apenas a uma *thread* ou a um pequeno sub-conjunto das *threads* presentes na execução. Por último, o GPU é um dispositivo *hardware* que foi feito para desenvolvimento de aplicações que trabalhem sobre grandes volumes de dados de maneira minimamente regular, por isto não estão preparados para lidarem com operações que exijam muitas alterações do fluxo de execução - *branching*. Isto faz com que o programador deva modificar o seu código para utilizar o menor número de operações como *if-then-else*, para evitar a mudança do fluxo de execução.

2.1.1.3 FPGAs

Um Field-Programmable Gate Array(FPGA) é na sua essência um conjunto de hardware programável. Estes dispositivos de harware são placas compostas por vários blocos de hardware que podem ter as ligações entre si configuradas através de Hardware Description Languages(HDL)[CB99].

Este tipo de hardware tem vindo a ter algum interesse por parte da comunidade científica, devido à flexibilidade deste tipo de dispositivos. As diferenças entre estes aparelhos e CPU ou GPU é facilmente notável, os FPGA têm uma estrutura mais complexa, para conseguir garantir a sua flexibilidade programável.

Algumas das vantagens que aqui podem ser identificadas é a rapidez do hardware em relação ao software. Qualquer coisa feita em software pode ser desenvolvida em hardware e ter um desempenho bastante melhor, mas a complexidade da construção muitas vezes não compensa esta abordagem.

Ainda assim existe alguma relutância em utilizar FPGA devido à sua complexidade e custos superiores ao CPU ou GPU convencionais, devido à sua complexidade. No entanto, convém referir que o custo deste tipo de hardware se pode tornar bastante atractivo quando o projecto que está implementado sobre o mesmo exige uma produção em massa. Este factor ganha um grande valor quando se fala em alterações sobre o projecto, visto que a arquitectura dos FPGA é modificável sem grandes custos, enquanto uma implementação com base em arquitecturas especificas por exigir mudanças completas de hardware.

2.1.1.4 Evoluções Recentes

Recentemente várias empresas têm feito apostas na área de processamento paralelo, sendo algumas das mais significativas apresentadas de seguida.

AMD Fusion As apostas mais recentes da AMD basearam-se no desenvolvimento de processadores que são compostos por vários *cores*, sendo alguns deles CPU *cores* e os restantes sejam GPU *cores*.

A arquitectura de uma AMD Fusion está representada na Figura 2.4. Como se pode ver este tipo de processador é composto por duas arquitecturas diferentes, neste caso é composta por 2 CPU *cores* e por 2 GPU *cores*. Estes *cores* partilham entre si um *bus* de memória, com a velocidade de 8 GB/s, e um sistema de memória DDR3. Cada *core* possui as suas *caches* sendo que cada L2 tem 512kB e cada L1 tem 32kB. Aqui aplica-se a regra em relação ao número de registos (blocos laranjas) e a capacidade de operação sobre vectores (blocos verdes), que aqui variam com o tipo de *core* a ser analisado.

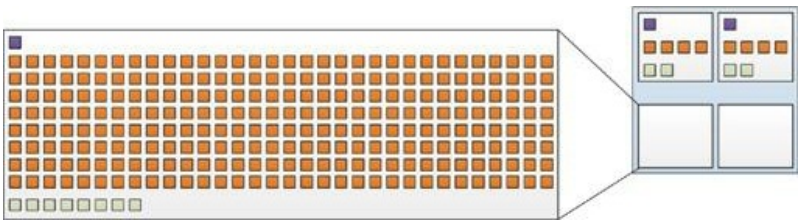

Figura 2.4: Arquitectura de uma AMD Fusion E-350. Retirado de [GKHM11]

Segundo a AMD esta é uma aposta no futuro. A evolução da computação exige processadores especializados em tarefas específicas, tal como já tem vindo a ser defendido por muitos no últimos tempos.

Com estes dois tipos de arquitecturas no mesmo *chip* a partilha de informação torna-se mais fácil e possivelmente mais rápida. Este tipo de processador também tem um menor consumo comparado a outros, visto que a união do CPU e GPU possibilita um melhor aproveitamento do *chip*. Algumas reservas a esta arquitectura têm sido levantadas relacionadas com a largura de banda de acesso à memória.

Intel MIC A abordagem da Intel é diferente da abordagem da AMD, possivelmente também afectada pela posição no mercado de processadores que cada empresa tem.

A AMD e a nVidia tem desenvolvido o hardware e para o programar é preciso recorrer por exemplo a OpenCL (2.1.2.2) ou CUDA (2.1.2.1). Enquanto isto acontece, a Intel tem desenvolvido o Intel Many Integrated Core(MIC), não um substituto para o típico CPU, mas uma espécie de *coprocessador* para o mesmo. Não para substituir o processador, mas para o auxiliar na acção de processamento.

Este tipo de dispositivo terá a capacidade de paralelizar código de uma aplicação sem grandes alterações, visto que o processador irá manter a compatibilidade com as instruções x86 actualmente utilizadas em muitas máquinas. Em [Int11] é descrita uma

experiência em que o processador MIC foi utilizado pela equipa OpenLab do European Organization for Nuclear Research(CERN) com um *benchmark* paralelo com bastante sucesso.

2.1.1.5 Conclusões

Os CPU Multi-core representam o hardware onde actualmente se encontra implementado o projecto. Tendo isto em conta, utilizar apenas estes tornou-se desinteressante, visto que os ganhos que poderiam ser obtidos apenas com o uso destes eram diminutos em relação a outras opções. No entanto este dispositivo hardware não pode nem vai ser excluído por completo do projecto.

Todos os outros dispositivos hardware apresentam características interessantes para implementação do projecto, no entanto optou-se por agora por GPU. Esta escolha-se deve-se principalmente ao melhor rácio custo/desempenho deste tipo de dispositivo, sendo o custo também um parâmetro a minimizar no projecto. O CPU irá servir como um distribuidor de carga, utilizando os recursos em GPU de maneira a maximizar o seu uso, beneficiando ao máximo o desempenho da aplicação e também irá executar tarefas que sejam mais propícias à execução neste tipo de processador.

Não foram escolhidas arquitecturas como a AMD Fusion ou o MIC visto que estes ainda são muito recentes no mercado. Os FPGA foram para já excluídos do projecto, mas podem vir a ter interesse se o projecto tomar uma grande escala e o seu valor possa ser diluído através da quantidade produzida.

2.1.2 Ambientes, Linguagens e Bibliotecas

As ferramentas para desenvolvimento de aplicações paralelas são imensas. As várias abordagens feitas podem ser classificadas nas seguintes categorias:

- **Novas Linguagens**. Algumas linguagens foram pensadas e desenvolvidas propositada e unicamente para que conseguissem lidar com o paralelismo de modo explícito e claro para o programadores;

- **Bibliotecas**. Uma abordagem mais comum é a utilização de bibliotecas. Já existem linguagens bastante poderosas e devidamente estruturadas, como exemplo C ou Java, portanto é mais fácil tentar utilizar bibliotecas que dêem algum nível de abstracção ao utilizador, sem lhe tirar controlo sobre o programa;

- **Paralelização pelo Compilador**. Esta é a abordagem ideal para o programador, porque consegue explorar o paralelismo de uma aplicação sem fazer grandes alterações ao seu código original ou preferencialmente até nenhumas. Foram criados compiladores que tentam explorar o paralelismo da aplicação fazendo a análise do seu código e explorando na maioria das vezes os ciclos presentes no código. Por vezes também são adicionadas algumas linhas de código ao programa sequencial,

que servem de directivas ao compilador do programa, tratadas caso este as consiga reconhecer, caso contrário irá tratá-las como comentários e compilar a versão original do código fonte.

Neste capítulo serão apresentadas e discutidas algumas abordagens a considerar neste projecto, tendo em conta que o hardware utilizado no projecto será CPU e GPU. Tendo em conta este aspecto serão estudadas algumas abordagens utilizando o GPU e outras abordagens que tentam lidar com a heterogeneidade das unidades de processamento das máquinas onde a aplicação será executada.

2.1.2.1 Disponíveis para GPU

Os GPUs foram desenvolvidos inicialmente para retirar o encargo que são as operações gráficas do CPU, tornando-se num tipo de processador virado para paralelismo do tipo SIMD, evidentemente ligado à matriz de *pixels* de uma imagem. Destes pormenores provém vários tipos de abordagem à programação para este tipo de hardware.

Passado Apesar de ainda faltar desenvolver muito trabalho sobre os modelos de programação em GPU, já muita evolução foi feita nesta área. Tal como já foi referido, o objectivo inicial dos GPU era trabalhar apenas com imagens, o que foi um obstáculo ultrapassado pela comunidade científica de maneira engenhosa.

Com as primitivas gráficas desenvolvidas para GPU, como o OpenGL [WNDS99], entre muitas outras, criar aplicações não gráficas era difícil. No entanto a comunidade começou a desenvolver este tipo de aplicações, com base em mapeamentos de matrizes em texturas, onde depois eram aplicadas funções definidas pelo programador. Este era um tipo de programação muito pouco amigável, felizmente já não é usual e hoje em dia já existem soluções mais razoáveis para programar em GPU.

CUDA O Compute Unified Device Architecture(CUDA[nVi08]) é uma arquitectura de computação paralela desenvolvida pela nVidia.

A programação em CUDA exige algum conhecimento por parte do utilizador em computação paralela e da própria arquitectura das placas da nVidia, visto que se faz um mapeamento directo das *threads* no GPU. O programador têm um nível de abstracção mínimo...

Qualquer programa em CUDA tem um esqueleto que segue quase sempre uma estrutura semelhante à seguinte:

- Definição do GPU onde o *kernel*, programa executado no GPU, vai ser executado;
- Possível *upload* de informação para GPU;
- Definição dos tamanhos dos *work_groups*, conjuntos de *threads* que partilham memória;

- Execução do *kernel*;
- Possibilidade de fazer uma sincronização entre o *host*, CPU, e o *target_device*, dispositivo onde é executado o *kernel*, normalmente o GPU;
- *Download* de alguma informação da memória do GPU;
- Limpeza das definições feitas no CUDA .

Qualquer programa escrito em CUDA tem pelo menos uma função *__global__ void*, que normalmente corresponde a uma função que deve ser executada em GPU - *kernel*. Esta função será depois chamada com uma parametrização directamente relacionada com o número de trabalhos a executar no GPU - *target_device*. O uso desta plataforma exige o recurso a um compilador específico, enquanto o resto do código do programa, ou seja, o que é executado em CPU continua a ser compilado por um compilador comum.

2.1.2.2 Heterogeneidade

Muitos são os que ficariam fortemente agradados, se fosse apresentado um modelo que conseguisse lidar com vários tipos de hardware, tanto CPU como GPU, entre outros. Este tipo de modelos já começaram a ser desenvolvidos e embora ainda não sejam utilizados em massa, muitos defendem que este vieram para ficar e mais tarde, ou mais cedo, irão afirmar-se na área da computação.

OpenCL O OpenCL[M+09] é uma plataforma que foi criada por um conjunto de empresas que decidiram desenvolver uma API capaz de abstrair as particularidades de CPUs, GPUs e outros processadores, permitindo a escrita de programas que executam num amplo leque de arquitecturas. O OpenCL é suportado por alguns gigantes da área da computação, designadamente AMD, Intel, nVidia, entre outros.

Esta API tem muitas semelhanças com o CUDA(2.1.2.1) da nVidia. Por exemplo, é necessário definir um *kernel* que irá correr num dispositivo seleccionado pelo programador à custa de invocação de primitivas do OpenCL. Os desempenhos em GPU da nVidia são consideravelmente melhores para programas desenvolvidos em CUDA[FVS11], do que em relação aos desenvolvidos em OpenCL, o que é razoável, visto que existe uma ligação directa na produção do hardware e do software em relação à programação em CUDA.

Um programa em OpenCL tem uma parte que é executada num CPU(*host*) que prepara a execução de programas (*kernels*) que correm noutro tipo de harware (*target*), por exemplo GPU. De seguida é apresentado o fluxo típico que um *host* percorre durante a execução de um programa desenvolvido com OpenCL.

- ***Query***[1] **aos Dispositivos**- Esta é a primeira fase de qualquer programa *host* em relação à computação com OpenCL. São feitas *queries* para descobrir os dispositivos

[1]Uma *query* consiste numa consulta feita a um conjunto de dados utilizando determinados filtros sobre os resultados retornados

que suportam OpenCL na máquina corrente e as suas respectivas características. Com base nos resultados destas *queries* é escolhido/escolhidos o/os *target_devices*;

- **Criação do Contexto**- A base de execução de um programa é maioritariamente composta pela criação de um contexto. Nesta estrutura de dados são definidos os aspectos mais importantes do programa, como os *target_devices*, o código do *kernel* e os objectos de memória de cada dispositivo;

- **Criação de Objectos de Memória**- Vários objectos de memória, tipicamente *buffers*, são criados para que a informação possa ser partilhada entre o *host* e o *target*. Durante a criação deste objectos, eles são associados a um contexto anteriormente criado;

- **Compilação e Criação de um *kernel***- Existe uma fase em que o código fonte deve ser carregado para memória. O que se faz é carregar esse mesmo código fonte para uma string, sendo posteriormente o código compilado de maneira a criar o próprio *kernel* que vai ser executado pelo *target_device*;

- **Lançamento de Comandos para a Fila**- É criada uma fila de comandos para coordenar a cooperação entre dispositivos. Com base nesta estrutura o *host* pode pedir ao *target* para executar determinado *kernel*, fazer alguma transferência de memória especifica ou até mesmo fazerem algum tipo de sincronização entre eles;

- **Sincronização de Comandos**- Muitas vezes é necessário que o *host* espere pelo fim de uma tarefa que está a ser executada pelo *target*, para tal existem pequenos indicadores associados às tarefas lançadas de maneira a ser feita uma sincronização entre os dispositivos. É típico encontrar este tipo de sincronização depois de lançar tarefas computacionalmente pesadas, para exigir a consistência dos dados em todo o espectro do programa.

- **Limpeza de Recursos**- Depois da execução do programa é sempre recomendável fazer uma limpeza à memória dos dispositivos, de maneira a que não fiquem em memória nenhuns dos objectos a ocupar espaço desnecessário.

Estas foram consideradas as funções mais importantes na preparação, compilação e execução de um *kernel* no *target_device*. No entanto existem muitas outras, dando outras opções, como por exemplo lançar um *kernel* num dispositivo com base nos seus binários em vez de ser no código fonte.

Uma função interessante que o OpenCL disponibiliza é uma *query* aos dispositivos de uma máquina. Esta *query* devolve informação descritiva de todos os dispositivos, designadamente o tipo do dispositivo, operações suportadas, versão do OpenCL suportada. Estas funções podem ser posteriormente utilizadas para ser feita uma escolha do dispositivo em que o *kernel* deve ser executado, para além de permitir definir parâmetros que podem optimizar a computação do programa.

O OpenCL mantém a heterogeneidade do hardware escondida do programador e apresenta-lhe uma abstracção baseada em unidades de trabalho, relacionada directamente com o número de núcleos do dispositivo utilizado na execução do código. Cada unidade de trabalho a executar corresponde a um *work-item*, em que cada um tem acesso a memória que está estruturada segunda uma hierarquia específica, apresentada na Figura 2.5.

Cada *work-item*, no OpenCL tem a sua memória privada, que não pode ser acedida por mais nenhum *work-item*. Estes *work-items* estão organizados em grupos, *Workgroup*, que partilham memória entre si. Também existe uma memória global do dispositivo, *Global/Constant Memory*, que é partilhada por todos os *work-item* do dispositivo. Por último existe a *Host Memory*, que basicamente é a memória controlada pelo dispositivo que lançou o *kernel* no dispositivo que contém os *work-item*.

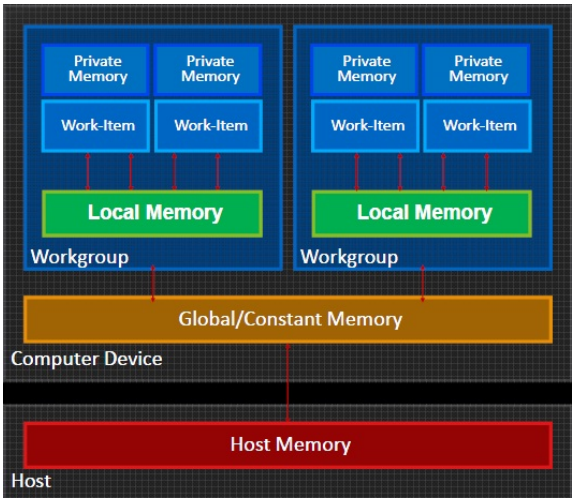

Figura 2.5: Modelo de memória no OpenCL. Retirado de [Mun08]

Os *work-itens* são mapeados no *target device* segundo uma grelha que para o caso mais comum actualmente pode ter entre 1 ou 3 dimensões. Este mapeamento pode-se tornar bastante útil para fazer uma boa divisão de trabalho entre as várias *threads* que vão ser executadas. Para lidar com isto cada *work-item* tem acesso a pequenas primitivas que lhes permitem saber em que zona da grelha estão, tanto a nível do seu grupo de *threads* como ao nível global, para além disto também são fornecidas primitivas para saber o tamanho destas dimensões, tanto a nível global como a nível de grupos.

O paralelismo no OpenCL pode ser de dois tipos, sendo estes os seguintes:

- **Paralelismo em Relação aos Dados**-Este é o modelo mais óbvio do sistema, basta para isto chamar um *kernel* e no código do mesmo, fazer análise a um subconjunto dos dados com base em alguma coisa, por exemplo o índice do *work_item*. Este é um tipo de paralelismo normalmente muito benéfico de explorar em dispositivos como GPU;

- **Paralelismo em Relação às Tarefas**-Este modelo pode ser recriado sobre esta ferramenta empilhando vários *kernels* numa *work_queue*. Este tipo de paralelismo é o que melhor se adapta aos CPUs *multi-core.*

Um *kernel* no OpenCL muitas vezes é escrito num ficheiro à parte, daquele que contém o código do *host* do programa, contrariamente ao CUDA(2.1.2.1) onde o *kernel* é uma simples função. O *kernel* segue a norma C99, com algumas limitações, tais como o uso de funções recursivas ou apontadores para funções.

Apesar destas limitações, algo que o código do *kernel* suporta são variáveis do tipo vector. Este tipo de dados servem para explorar o paralelismo das operações SIMD, permitidas no *target_device*, dependente do tipo de dispositivo.

Sendo o OpenCL uma plataforma para explorar paralelismo, possui métodos de sincronização, tanto a nível do *kernel* como do *host*. Mas existem algumas limitações, tais como a impossibilidade de sincronizar dois *work_items* contidos em *work_groups* diferentes de maneira clara. Para que esta sincronização seja possível é necessário recorrer a operações atómicas sobre a memória global do dispositivo, podendo ser muito difícil controlar este tipo de operações.

Outras Abordagens Uma abordagem que conseguisse lidar com a heterogeneidade do hardware seria algo que qualquer programador estaria interessado em experimentar. Visto isto, tem havido mais tentativas para lidar com este factor para além do OpenCL. De seguida são apresentadas e analisadas brevemente algumas abordagens que tiveram origem similar ao OpenCL, mas que tentam dar uma maior nível de abstracção ao utilizador das mesma.

PGI Accelerator O PGI Accelerator é uma ferramenta, apresentada em [Wol10], que tem como umas das suas principais características o uso de directivas, similares às do OpenMP[DM98]. Este projecto faz parte do trabalho desenvolvido pelo grupo "The Portland Group", membro do consórcio que trabalha no desenvolvimento do OpenACC(2.1.2.2).

Esta ferramenta está concebida, até à data, para efectivamente trabalhar sobre três linguagens, o C, C++ e Fortran. Aqui consegue-se um grande nível de abstracção da heterogeneidade do hardware e o espectro de acção desta ferramenta é bastante largo, conseguindo trabalhar sobre todo o hardware baseado na arquitectura x86 ou que consiga suportar CUDA(2.1.2.1), mais especificamente os GPU da nVidia.

Este tipo de modelo é bastante atractivo, no entanto perde por restringir o seu uso comercialmente, apesar de se poder considerar o antepassado do OpenACC, que poderá vir a tornar-se numa norma na área.

OpenACC A programação com base em directivas tem bastantes apoiantes e embora por vezes o desempenho não seja o óptimo, é defendido por muitos que tal perda é

irrelevante quando se considera a facilidade de programação. Como tal, foi apresentada uma nova proposta de programação com base em directivas para GPU, de nome OpenACC, apresentado em [CAP11], desenvolvida por um grupo de empresas com grandes capacidades no mundo da computação e da compilação.

Esta proposta apenas está a ser desenvolvida para GPU, mas baseia-se na interpretação de código C, C++ ou Fortran que depois é carregado para a GPU. Tendo isto em atenção será possível utilizar o OpenACC em conjunto com abordagens dirigidas ao CPU, como por exemplo o OpenMP.

Já é apontado por muitos como um modelo heterogéneo de programação paralela que pode vir a ser líder nesta área.

2.1.2.3 Conclusões

Para permitir acomodar diferentes tipos de dispositivos (GPU, tanto AMD como nVidia, e eventualmente outros aceleradores), o OpenCL vai ser usado como plataforma de paralelização. Esta escolha deve-se principalmente ao grande desenvolvimento, em termos de projectos, que já existe sobre o OpenCL. Também existe muito trabalho feito em CUDA, mas essa plataforma foi rejeitada para não nos prender às tecnologias somente associadas a uma marca de GPU, nVidia.

Como comentário final vale a pena dizer que os ambientes disponíveis para programação de GPU parecem caber em duas categorias:

Alto Nível Permitem ao programador uma especificação relativamente simples da estratégia de paralelização, mas no caso das GPU os níveis de desempenho atingidos são menores do que aqueles que o hardware permitiria.

Baixo Nível Expõem ao programador todos os detalhes do hardware nomeadamente a hierarquia de memória, o que permite atingir níveis de desempenho mais próximos do potencial do hardware; a experiência mostra que a obtenção desses níveis elevados de desempenho se faz, muitas vezes, à custa de um longo processo de ajuste do algoritmo.

A estas dificuldades acresce a ausência de metodologias que permitam uma fácil paralelização de aplicações que não encaixem no modelo SIMD. Nos trabalhos conducentes a esta dissertação os algoritmos a paralelizar não encaixam neste mesmo modelo.

2.1.3 Grafos em GPU

Foi concluído que o processo de criação do grafo que representa as ligações entre n-gramas é a operação mais demorada do classificador. A a criação do grafo nos algoritmos estudados é dinâmica e deriva de comparações sintácticas entre os distintos n-gramas de um documento. As características específicas desta operação obrigaram a um estudo e análise cuidada das estruturas para grafos desenvolvidas e utilizadas em vários tipos de

hardware; este estudo teve como objectivo concluir quais seriam as estruturas que melhor se adaptariam ao comportamento esperado do algoritmos que trabalham sobre o grafo.

A criação de grafos em GPU é um assunto que já se encontra relativamente estudado [LH, NLKB11], no entanto a pesquisa relevante nesta área tem-se centrado no caso em que os arcos dos grafos são gerados aleatoriamente.

Foi necessário fazer o levantamento de várias estruturas utilizadas para representar grafos que serão de seguidamente apresentadas e avaliadas nos termos deste projecto. Convém ter em conta que o grafo criado pelo processo analisado nesta dissertação tem uma quantidade pequena de arcos em relação a um grafo completo, o que também aproxima em muito o material apresentado a representações de matrizes esparsas. Uma outra característica sobre os grafos criados que deve ser tida em conta na análise das estruturas apresentadas é que, no nosso caso há uma grande variação no número de arcos associados a um nó. Ou seja, enquanto um nó pode ter ligações a quase todos os outros é bastante provável que exista um outro nó com um diminuto número de arcos em relação ao número de nós do grafo.

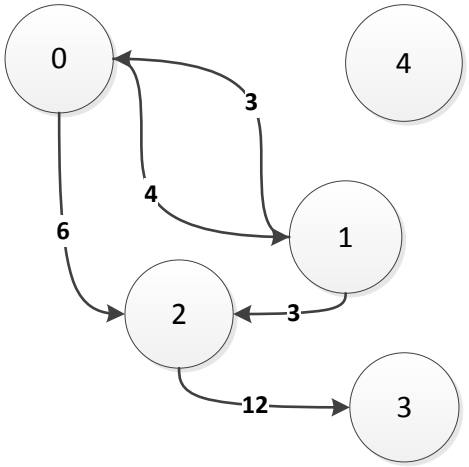

Figura 2.6: Grafo exemplificativo

Um grafo pode ser definido como um conjunto de nós e um conjunto de arcos, sendo que cada arco liga dois nós. Ao longo da apresentação das questões relacionadas com este tema serão apresentados alguns exemplos, todos eles serão demonstrações de formatos específicos para codificar o grafo mostrado na Figura 2.6.

As estruturas apresentadas são as que são utilizadas em maior escala, tanto em CPU como em GPU, e as que se adaptam melhor à criação de grafos. No entanto existem mais algumas estruturas que não serão aqui analisadas devido à sua natureza que as torna pouco relevantes para este problema, de onde se destaca JDS[DLN+94] e SKS[DLN+94, Saa94].

No final serão apresentadas algumas observações sobre quais as estruturas identificadas como mais benéficas para este problema em particular, o porquê de serem as mais benéficas, sendo apresentadas mais à frente na dissertação as modificações que foram feitas a estas mesmas estruturas para melhor se adaptarem ao problema em causa.

2.1.3.1 Formato Denso ou Matriz de Adjacências

Este é o formato de armazenamento de um grafo mais básico possível. Numa matriz M com dimensões nxn definimos que existe um arco do vértice i para o vértice j se $M(i,j) <> 0$, sendo que o peso do arco será o valor presente em $M(i,j)$, tal como está explícito em Def. 1.

Definição 1 $\forall(i < n)\forall(j < n) \begin{cases} M(i,j) = 0 & \textit{Não existe arco} \\ M(i,j) <> 0 & \textit{Arco com peso M(i,j) do nó } i \textit{ para o nó } j \end{cases}$

$$\begin{bmatrix} 0 & 3 & 6 & 0 & 0 \\ 4 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 12 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Figura 2.7: Estrutura de uma Matriz de Adjacências

Na Figura 2.7 pode-se ver um mapeamento entre o grafo mostrado na Figura 2.6 e uma destas estruturas. O mapeamento de um grafo neste tipo de estrutura é bastante directo, no entanto pode ser um formato custoso do ponto de vista da memória. A análise desta estrutura por *threads* paralelas também é de fácil implementação, visto que a estrutura pode ser facilmente repartida, utilizando linhas ou colunas, traduzindo-se numa partição por sucessores ou antecessores.

2.1.3.2 *Coordinate List (COO)*

Este tipo de estrutura[Saa94, DLN+94, KMSM12] guarda uma lista de tuplos(linha, coluna, peso do arco). Basicamente cada coluna da matriz desta estrutura representa um arco enquanto cada linha representa uma característica especifica de um conjunto de arcos.

Este formato é de uma simplicidade imensa e é muito bom para a construção incremental de um grafo. Uma figura exemplificativa deste tipo de estrutura está ilustrada na Figura 2.8; esta figura mapeia o grafo mostrado na Figura 2.6.

Para adaptar esta estrutura especificamente para GPU, usando OpenCL, basta alocar previamente os recursos para esta estrutura e ter o cuidado de não exceder os limites dos recursos alocados, devido à inexistência de alocação dinâmica de memória. Isto pelo menos para a abordagem mais simplista.

Origem	0	0	1	1	2
Destino	1	2	0	2	3
Peso	3	6	4	3	12

Figura 2.8: Estrutura de uma COO

2.1.3.3 Lista de Adjacências(*List of Lists-LIL*)

Ao contrário da matriz de adjacências que guarda as relações que existem entre todos os nós, a LIL apenas guarda as relações que se traduzem em arcos presentes no grafo. Uma lista principal contém n listas, sendo que cada uma destas sub-listas guarda a lista de sucessores, ou antecessores do nó que representam.

Tal como está representado na Figura 2.9 existe uma lista que é atribuída a cada nó, neste caso 5 nós. Cada uma dessas lista mostra os antecessores ou sucessores do nó e o respectivo peso desse arco, nesta figura são mostrados os sucessores de cada nó. Para além disto é mantida numa lista um registo sobre o número de arcos que cada lista contém.

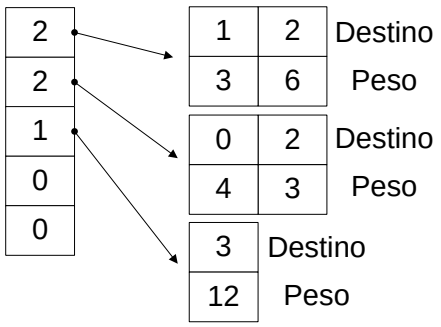

Figura 2.9: Estrutura de uma LIL

Como este tipo de estrutura é baseado numa lista será mais facilmente relacionado com técnicas que utilizem memória dinâmica. No entanto podemos adaptá-la ao uso de memória estática, desde que também estejamos dispostos a desperdiçar alguma memória adaptando esta técnica a arrays.

2.1.3.4 *Compressed Sparse Row (CSR ou CRS)*

O formato CSR[Saa94, DLN+94, KMSM12] é largamente utilizado em projectos que trabalham com matrizes esparsas em GPU.

Este tipo de estrutura permite uma fácil partição do grafo para análise paralela do mesmo e um bom aproveitamento da memória do dispositivo. Com este tipo de estrutura é trivial descobrir todos os sucessores de cada nó.

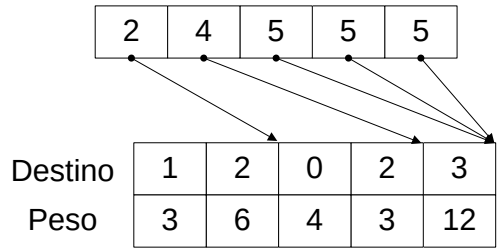

Figura 2.10: Estrutura de uma CSR

Tal como pode ser visto na Figura 2.10, existem duas listas, sendo a maior a que contém a informação sobre todas as sucessões de nós que existem no grafo. A primeira lista serve para relacionar a origem do arco com a sucessões que lhe pertencem.

Basicamente a primeira lista dá o índice da lista de sucessões onde começam a ser guardados os arcos relativos a esse nó. No exemplo apresentado significa que o nó 0 terá arcos desde o índice 0 até ao 2(valor da 1ªlista na posição 0) exclusive na lista de sucessões, o nó 1 terá arcos na lista de sucessões desde o índice 2(valor da 1ªlista na posição 0) até ao 4(valor da 1ªlista na posição 1) exclusive e por aí diante.

2.1.3.5 *Compressed Sparse Column (CSC ou CCS)*

Este formato[Saa94] é uma variação do CSR que em vez de fazer a referência com base em linhas da matriz de adjacências, faz com base nas colunas. Ao contrário do formato CSR, este formato permite uma leitura trivial de todos os antecessores de cada nó invés dos sucessores. Um exemplo desta implementação é mostrado na Figura 2.11.

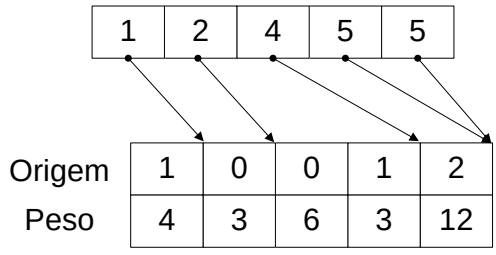

Figura 2.11: Estrutura de uma CSC

A primeira lista dá o índice da lista de antecessores onde começam a ser guardados os arcos relativos a esse nó. No exemplo apresentado significa que o nó 0 terá arcos desde o índice 0 até ao 1(valor da 1ªlista na posição 0) exclusive na lista de antecessores, o nó 1 terá arcos na lista de sucessões desde o índice 1(valor da 1ªlista na posição 0) até ao 2(valor da 1ªlista na posição 1) exclusive e por aí diante.

2.1.3.6 *ELLPACK (ELL)*

O formato ELLPACK[Saa94] tenta representar uma matriz esparsa utilizando duas matrizes auxiliares. Estas duas matrizes têm n linhas, sendo cada uma associada a um nó do grafo, e um número de colunas igual ao maior número de sucessores que um nó tem no grafo

Se o número de arcos presentes no grafo for demasiado elevado este formato pode consumir mais memória do que uma matriz de adjacências.

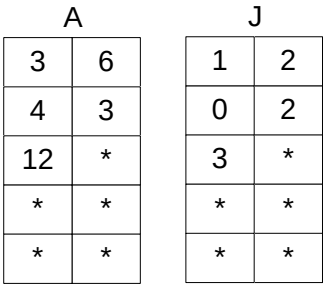

Figura 2.12: Estrutura de uma ELL

As duas matrizes que existem têm a mesma função, mas mostram informação diferente. A matriz A mostra o peso de um arco e a matriz J mostra o destino ou origem desse mesmo arco, o sucessor no caso da Figura 2.12.

2.1.3.7 Conclusões/Observações

Grande parte dos formatos apresentados centram-se na compressão da informação de um grafo, onde se destacam os mais largamente utilizados em GPU, que são o CSC, CSR e ELL. Este não é o factor que mais nos preocupa, embora se deva sempre ter em mente que a memória é escassa e preciosa. Segundo estas características os formatos referidos anteriormente tiveram de ser excluídos, porque as suas técnicas de compressão dificultam a criação de um grafo concorrentemente.

Qualquer um dos outros formatos apresentados poderiam ser aproveitados na criação de um grafo de forma concorrente. Numa fase intermédia do projecto isto permitiu algumas experimentações e testes interessantes com estas estruturas. No entanto a fase final do projecto exigiria a utilização de um algoritmo que é uma variação do conhecido algoritmo HITS[DHH$^+$02]; este algoritmo exige iterações sobre os vários antecessores e sucessores de cada nó. Este ponto fez com que a COO fosse completamente descartada para utilização neste problema, visto que os arcos contidos nesta lista não seguem nenhum tipo de ordenação específico, logo não permitindo a iteração directa sobre os antecessores ou sucessores de cada nó. Posteriormente a matriz de adjacências também foi excluída, visto que a iteração sobre os arcos de um nó seria algo pouco adaptável à arquitectura de um GPU,visto que o grafo tem poucos arcos.

Assim ficou decidido utilizar no projecto a estrutura LIL adaptada sobre matrizes, para melhor se adaptar a GPU. Esta estrutura iria permitir a fácil e eficiente iteração sobre ligações de cada nó, sem prejudicar em demasia a memória utilizada e permitindo a tão desejada criação de um grafo concorrentemente. O espaço em memória gasto seria bastante superior ao necessário para guardar a informação sobre o grafo criado, no entanto este factor não é preocupante, visto que a memória em GPU não é uma limitação do projecto.

2.2 Solução/Implementação

Este capítulo encontra-se omitido devido a questões de confidencialidade empresarial.

2.3 Resultados e Análise

A análise à implementação concebida baseia-se num conjunto de testes, onde existe uma grande diversidade no tamanho dos vários pedidos. A avaliação consistiu basicamente numa análise aos tempos de execução de cada implementação quando confrontados com o mesmo cenário.

Os vários resultados apresentados são baseados em médias retirados de várias repetições dos mesmos testes. Esta opção tem como objectivo diminuir o ruído nos testes provenientes de factores exteriores ao mesmo. Não foram apresentadas barras de erro nos gráficos apresentados, porque o desvio nos valores dos vários testes em relação à média obtida é diminuto em relação às grandezas que são analisadas.

Deve ser referido que a versão em CPU apenas utiliza um núcleo do processador, visto que em 3.3 estes serão utilizados para o processamento de vários pedidos concorrentemente.

De seguida são apresentadas as configurações de hardware e software utilizadas nestes testes.

- **Contexto de Experimentação** As experiências foram feitas num PC, que tem como hardware base: um processador Xeon E5504 com 4 núcleos; 12 Gbytes RAM; um acelerador(GPU).

  O acelerador que a máquina possuí é um nVidia GTX680 (Kepler), que consiste num GPGPU com 1536 núcleos CUDA, com 2 Gbytes de memória.

  Relativamente ao software utilizado, este consistiu na plataforma OpenCL sobre o sistema operativo Linux na distribuição Ubuntu, versão 10.04.4, kernel 2.6.32-43; a versão do gcc usada é a 4.4.3. Convêm também referir que as máquinas utilizadas funcionam num ambiente partilhado entre múltiplos utilizadores, o que pode prejudicar ligeiramente o resultado de alguns testes, independentemente da versão a ser testada.

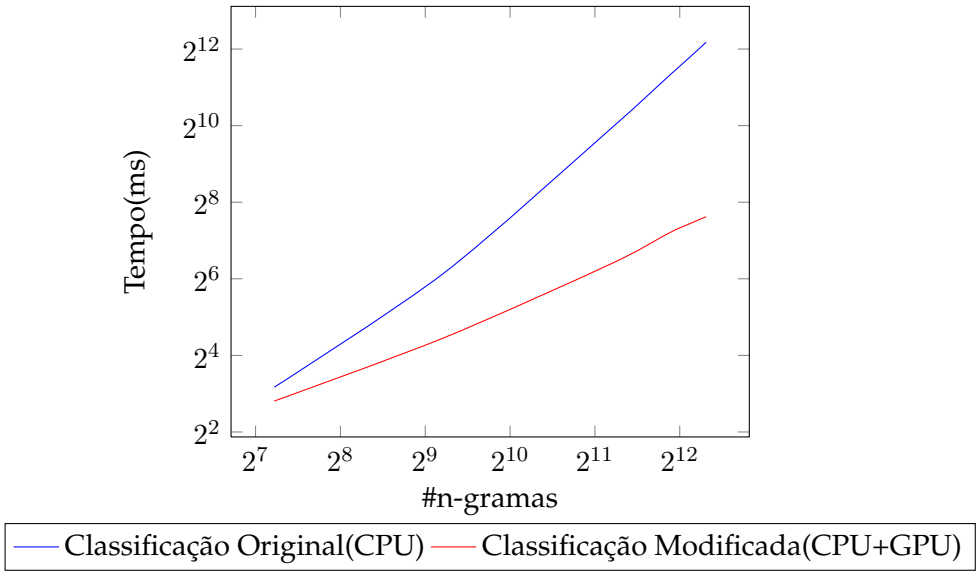

Figura 2.13: Tempos de Execução de uma Classificação

O principal interesse de comparação foca-se sobre o tempo de uma classificação completa, no entanto também será apresentado os tempos em relação apenas ao algoritmo de extracção de *features*, visto que este foi o único módulo alterado ao longo do trabalho.

Embora não seja o único factor relevante para o tempo que uma classificação demora, o número de n-gramas distintos extraídos quando se chega ao algoritmo WHAKA é o factor que mais peso tem neste resultado. Como tal foram criados alguns gráficos que mostram o tempo de classificação em função deste factor, que nunca excede o valor 6000, visto que os documentos a serem classificados são páginas web e quase nunca excedem este limite num contexto real de utilização do projecto.

Os principais gráficos desta avaliação são os que estão representados na Figura 2.13 e na Figura 2.14. Estes gráficos mostram os tempos em relação à classificações completas, ou seja que passam por todo o processo apresentado no Capítulo 1.

Podemos ver que independentemente da versão do programa o tempo de classificação aumenta de forma regular em função do aumento do número de n-gramas do documento a ser classificado. No entanto em relação à versão original, pode-se ver que o crescimento deste tempo ocorre mais lentamente para a versão modificado do que para original.

O gráfico representado na Figura 2.14 faz a relação entre os tempos de classificação das duas versões do programa, ou seja, demonstra o *speedup* da versão modificada em relação à versão original.

Pela tendência das linha do gráfico também podemos dizer que provavelmente para diferentes configurações de hardware e para problemas com tamanhos pequenos, neste caso com menos de 128 n-gramas, a versão original pode tornar-se mais rápida do que a versão modificada. Isto deve-se à dimensão pequena do trabalho que impede uma divisão do mesmo pelos vários recursos disponibilizados pelo GPU.

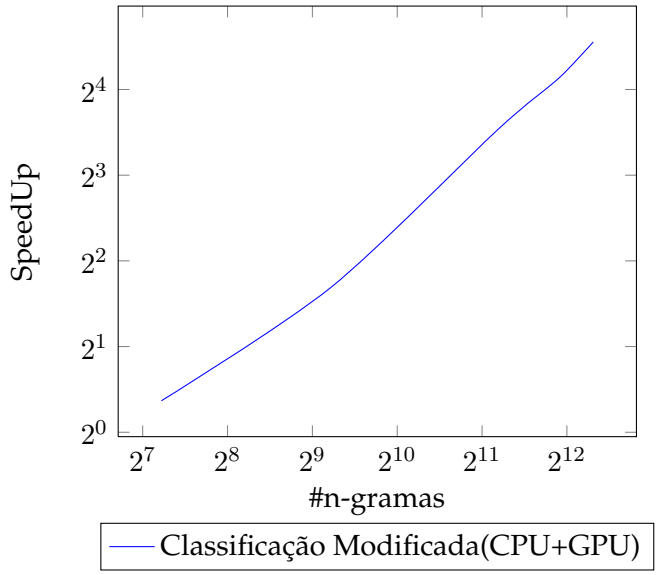

Figura 2.14: SpeedUp de uma Classificação

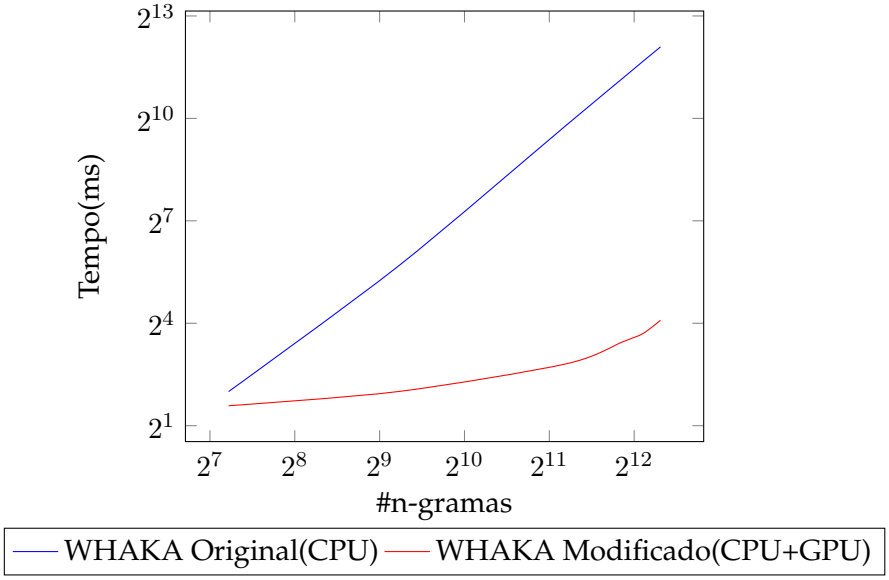

Figura 2.15: Tempos de Execução do Algoritmo WHAKA

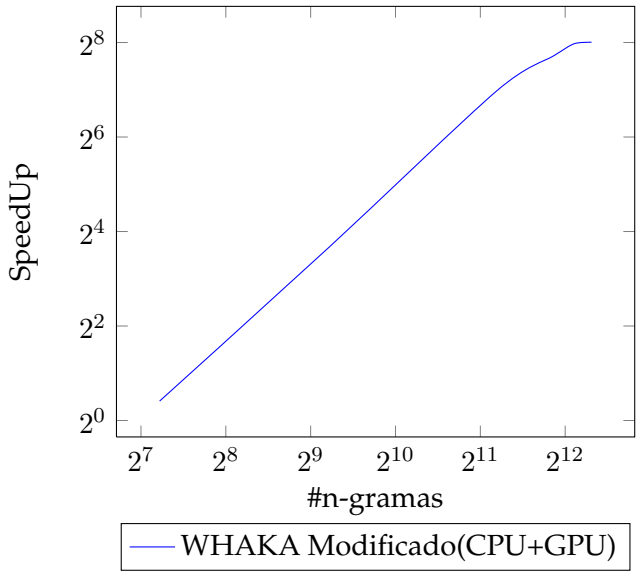

Figura 2.16: SpeedUp do Algoritmo WHAKA

Os gráficos representados na Figura 2.15 e na Figura 2.16 têm maior interesse, porque analisam os ganhos obtidos da transformação do único módulo modificado durante todo o projecto. Tal como seria de esperar, nestes dados apresentados como gráficos a versão modificada apresenta óptimos resultados.

Apesar dos resultados em relação a uma classificação geral também sejam muito bons, não são tão expressivos como estes visto que os ganhos obtidos com as modificações do algoritmo WHAKA se vão dissipando nos tempos de execução dos outros componentes da aplicação.

3

Aceleração do Processamento de Conjuntos de Pedidos

Tendo em vista o objectivo final da aplicação, tivemos de ter em conta a aceleração de um ponto de vista um pouco diferente. O objectivo final da aplicação será manter-se activa num servidor a atender vários pedidos de classificação. Embora seja interessante atender cada pedido o mais rápido possível, o mais importante torna-se atender o maior número possível de pedidos num determinado espaço de tempo.

Espera-se que esta aplicação receba largos milhares de pedidos por hora, nós queremos atender este conjunto o mais depressa possível. No entanto não podemos prejudicar em demasia a velocidade de atendimento de um pedido. Para satisfazer este tipo de requisitos foi desenvolvido um módulo capaz de aproveitar todos os recursos do hardware utilizado, CPU e GPU, aumentado o *throughput* da aplicação no seu âmbito global.

Originalmente o projecto tratava tudo com a abordagem mais comum possível. Existia um processo a atender pedidos e estes são adicionados a uma fila de onde várias *threads* vão tirar trabalho, após processar o mesmo enviam os seus resultados. É uma abordagem simples e eficaz, mas padece das limitações do processador que utiliza, o CPU. O número de processos de classificação em execução num determinado momento encontra-se sempre limitado pelo número de núcleos do processador que está a ser utilizado.

De seguida são apresentadas algumas secções em que se mostra o trabalho relacionado com este tipo de abordagem, a solução escolhida e o porquê de determinadas escolhas desta solução e uma última secção composta por uma análise e discussão sobre o produto final criado.

3.1 Trabalho Relacionado

Serão aqui apresentados alguns projectos que seguiram técnicas de paralelização em GPU similares à proposta para resolver o problema apresentado neste capítulo. O problema aqui apresentado será em relação à baixa utilização dos recursos disponibilizados por GPU e de seguida serão apresentadas duas técnicas que tem como objectivo final resolver este problemas.

O processamento de vários pedidos em paralelo - *batch/bulk* - será aqui avaliado e serão discutidas as suas vantagens e defeitos e para além disso quais são os tipos de problemas aos quais melhor se adapta este tipo de técnica.

A última secção deste capítulo tem como objectivo apresentar uma plataforma criada em CUDA com objectivos muito similares aos apresentados nesta fase deste trabalho. Embora seja uma plataforma que apenas tenta possibilitar a plataforma CUDA de um esquema paralelo entre tarefas, também pode ser facilmente adaptada ao caso deste projecto, em que esta faceta é explorada em conjunto com o paralelismo entre processos, classificações no nosso caso.

3.1.1 Processamento em *Bulk/Batch*

As aplicações que até aos dias de hoje têm sido aceleradas em GPU, podem ser divididas em dois grandes grupos:

- **1 Processamento**- Quando uma aplicação tem tempos execução grandes, mas este apenas representa uma instância do problema. Este tipo de contexto é muito comum, algumas das aplicações que se encaixam neste perfil são aplicações sobre previsão meteorológica ou previsões financeiras a nível da banca. Normalmente a abordagem para este tipo de problema baseia-se em acelerar um processo em geral;

- **Processamento em *Bulk/Batch***- Muitas vezes o tempo de processamento de uma instância de uma aplicação é pequeno, quando comparado com o tempo necessário à criação e terminação do processo que a executa, a sua importação para GPU parece inapropriado. No entanto um outro tipo de investimento pode ser feito, *throughput* em vez de aceleração. Nestes casos normalmente a aplicação exige o processamento de várias instâncias, por vezes até em simultâneo, de um problema. Aqui podemos ao invés de simplesmente acelerar a aplicação pensar em modificá-la para conseguir responder a mais instâncias do problema em simultâneo aproveitando assim a capacidade de processamento do GPU. No fim de contas o processo, conjunto de problemas, será mais rápido já que com aumento do *throughput*, também se consegue diminuir o tempo de espera de várias instâncias do problema e assim diminuir o tempo total de execução;

O tipo de aplicações que nos interessam e que são similares à criada nesta fase do projecto encontra-se no segundo grupo apresentado.

Vários testemunhos sobre ganhos relevantes neste tipo de aplicações têm sido relatados pela comunidade cientifica, como em [NI09, MZZ+10]. Nestes casos o trabalho foi feita sobre pequenas tarefas que tinham como objectivo algum tipo de análise sobre pacotes que circulavam na rede, ou seja, aplicações que se aplicam ao perfil anteriormente definido, tarefas pequenas, mas feitas em grande quantidade.

Também existe trabalho similar por parte de aplicações que tem como objectivo implementar pesquisas em grandes base de dados com recurso a GPU, tal como é apresentado em [HLY+09, WZA+09].

Nos 4 casos anteriormente referidos a única solução que tornaria aceitável o recurso a GPU estava no processamento em *bulk/batch* de maneira a aumentar o *throughput* do respectivo projecto. Em qualquer um dos casos apresentados, o tempo de processamento de um só elemento(1 pacote ou 1 query) era diminuto, o que tornaria o processamento de cada um em GPU algo dispendioso, visto que dificilmente se teria um bom aproveitamento dos recursos utilizados.

Para contornar o que poderia ser um mau aproveitamento dos recursos foram criadas estruturas que conseguiam mapear vários elementos que deviam ser processados pelos vários recursos do GPU de maneira a ter um bom aproveitamento dos seus recursos. Tal como seria esperado, o processamento de um único elemento mantinha-se similar a um processamento sem *bulk/batch*. No entanto o número de elementos processados por intervalo de tempo tem tendência a crescer em todos os projectos citados.

3.1.2 Paralelismo entre Tarefas

Desde de sempre, o GPU era e ainda é classificado como um processador facilmente adaptável a problemas paralelos em relação aos dados.

No entanto, o paralelismo em relação a tarefas, apenas começou a ser abordado por plataformas como o CUDA ou OpenCL recentemente. Este é um grande desafio, porque tal como foi anteriormente referido este tipo de hardware pode ser classificado como SIMD, embora muito mais haja para ser tido em relação a isto. A execução de dois programas, *kernels*, diferentes simultaneamente em GPU ainda está sujeita a muitas limitações e é de difícil exploração em plataformas para programação generalizada como é o caso do CUDA ou OpenCL de maneira clara e simples.

Apesar disto, a comunidade científica tem forçado a entrada deste modelo de programação neste tipo de hardware, sendo um dos esforços mais significativos e interessantes nesta área apresentado em [GGHS09]. Este projecto mostra as vantagens que podem existir em fundir dois *kernels* que vão ser executados em GPU, com o recurso a apenas uma simples cláusula *if-then-else* e alguma factorizarão de código. Os ganhos variam conforme a natureza dos *kernels* fundidos, *I/O bound* ou *compute bound*, e com o tamanho dos problemas devido à diferença no uso dos recursos e aproveitamento dos mesmos.

Aqui é apresentada a fusão de diferentes *kernels* que não têm qualquer relação entre si, embora isto possa ser especializado para o nosso caso. Este aspecto exige um controlo

rígido da memória exigida pelos diferentes *kernels*, para que as requisições de memória não excedam a que esteja disponível em GPU.

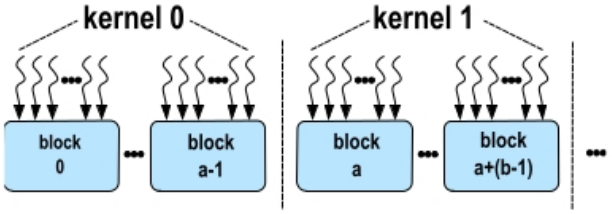

Figura 3.1: Atribuição de *threads* a *kernels* fundidos. Retirado de [GGHS09]

O uso desta técnica não tem problemas de divergência do fluxo de processamento dentro de *warps*, visto que a divisão de trabalho é feita por cada grupo de *threads*, tal como está ilustrado na Figura 3.1. Este tipo de divisão evita este problema, aumentando a eficiência do uso dos recursos do GPU.

3.2 Solução/Implementação

O trabalho desenvolvido nesta fase tinha como objectivo criar um projecto capaz de usar os recursos do GPU com grande aproveitamento e sempre na sua totalidade, mas continuando a dar um nível de abstracção razoável sobre estes recursos em uso.

Segundo os objectivos propostos foi desenvolvida uma arquitectura que está exemplificada na Figura 3.2. Tal como é demonstrado continuamos a ter módulos que são executados antes da extracção de *features* e módulos que são executados depois, seguindo o mesmo fluxo de execução do projecto original. O módulo WHAKA, interno ao componente anteriormente referido, é que sofreu uma complexa reestruturação, nesta pode-se começar por salientar a divisão do mesmo em duas implementações continuando uma a utilizar CPU, enquanto a segunda iria utiliza GPU. A razão para estas duas implementações deve-se ao facto de existirem casos em que cada tipo de hardware tinha benefícios em relação ao outro, isto é mais notório em relação a exemplos em que o *overhead* do uso de GPU se sobrepõe ao ganho de aceleração no processamento de um pedido. Ou seja, na classificação de conteúdos pequenos seria benéfico utilizar CPU e para conteúdos de dimensão maior os benefícios estariam no uso de GPU, sendo os benefícios maiores com o aumento da dimensão do problema.

O valor que define o limite entre a utilização dos diferentes processadores terá de ser obtido por experimentação e ficará representado no novo módulo IGPUServer, que será apresentado de seguida dando mais alguns detalhes sobre este assunto. Este valor deve ser obtido com base nos tempos de execução do algoritmo em CPU e em GPU, ficando o limite para escolha do hardware de execução do algoritmo perto do número médio de n-gramas de um pedido. Este limite será o número de n-gramas do pedido em que os dois tipos de arquitectura têm o mesmo tempo de processamento.

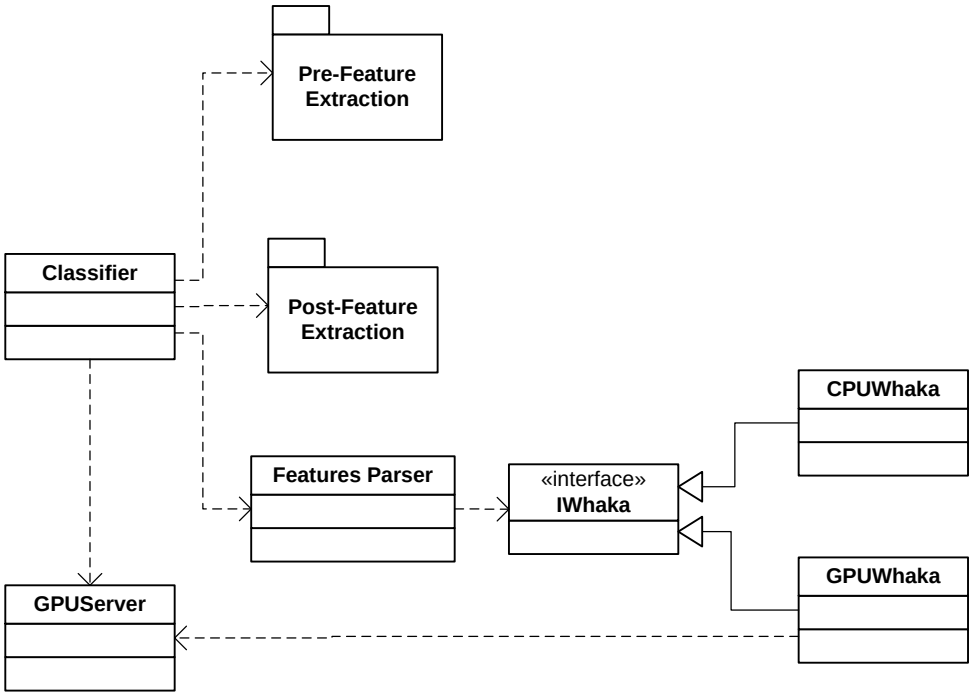

Figura 3.2: Arquitectura da Solução

A solução deste problema foi concebida de maneira a ser de simples utilização para o processo que utiliza o GPU, preferencialmente sem ter noção do mesmo.

A inicialização das ferramentas deste módulo acrescentam um *overhead* ao programa em si, no entanto este é desprezado visto que esta inicialização apenas acontece um vez e não tem qualquer influência sobre qualquer pedido de classificação.

Foi definida uma interface básica para o módulo GPUServer que está ilustrada no excerto mostrado de seguida:

```
1  interface ÍGPUServer{
2
3    /*
4    * Tenta reservar recursos para o processo, se não tiver suficientes
5    * devolve -1, caso contrário devolve um id que identifica o processo
6    * no servidor
7    */
8    int registRequest(list<uint> bucketSizes);
9
10   /*
11   * Após a tentativa bem sucedida de registo no servidor o processo
12   * que invocou essa chamada deve submeter alguma da sua informação
13   * de controlo utilizando esta chamada
14   */
15   void flushÍnfo(int requestÍd, requestÍnfo r);
```

```
16
17    /*
18    * Íncrementalmente o processo de classificação deve ir adicionando
19    * n-gramas no seu registo do servidor.
20    * Esta chamada pode desencadear tarefas em GPU, sem o conhecimento
21    * do processo que a invocou
22    */
23    void addNgram(int requestÍd, ngram g);
24
25    /*
26    * Para finalizar, o processo de classificação pede os resultados
27    * ao servidor, dando-lhe a indicação que diz onde a informação
28    * deve ser guardada.
29    * Caso as tarefas em GPU deste processo ainda não tenham
30    * sido desencadeadas serão certamente com esta chamada
31    */
32    void getResults(int requestÍd, float* comp1, float* comp2);
33  }
```

Esta interface é utilizada pelos processos de classificação e qualquer uma delas pode desencadear operações em GPU sem o conhecimento desse mesmo processo. As tarefas apenas são submetidas para GPU quando todas as que a precedem funcionalmente foram executadas e todos os *uploads* de informação necessários à mesma também já foram terminados. Apenas os processos de classificação podem fazer *upload* de informação privada ao seu pedido, existindo depois uma colaboração para sincronizar estes vários eventos.

Existe uma *thread* em CPU que periodicamente executa em GPU as tarefas submetidas pelas *threads* que são pedidos de classificação. Quando não são submetidas tarefas durante um determinado período de tempo, a *thread* principal fica num estado de espera que apenas é quebrado quando é identificada uma nova submissão de tarefas à fila de tarefas.

No início do projecto apenas existia um módulo Whaka, que agora definimos como CPUWhaka. Este pequeno módulo era composto por uma interface e pela sua implementação, nesta fase a interface foi generalizada e foram criadas duas implementações, sendo a primeira baseada na solução original do problema e a segundo num esquema de trabalho com o módulo que serve de abstracção do GPU.

O módulo IWhaka tem a sua interface representa no seguinte excerto de código:

```
1     /*
2     * Adiciona um n-grama ao componente whaka e define de imediato
3     * algumas características do mesmo
4     */
5     void addNode(N-gram n, float initialValue, Sentences[] sents)
6
7     /*
8     * Força a execução do algoritmo de dispersão de pesos, dando
9     * a certeza de que não serão submetidos mais n-gramas a este
```

```
10   * componente
11   */
12   void compute(uint iterations)
13
14   /*
15   * Devolve o conjunto de nós resultantes da submissão do vários
16   * n-gramas, com informação sobre os pesos e constituição de cada
17   * um. Esta informação serve para averiguação da relevância dos
18   * vários n-gramas no documento em análise
19   */
20   Node[] getNodes()
```

Este módulo mantém exactamente as chamadas que existiam neste tipo de interface no projecto inicial. Esta forte correlação entre as várias versões facilita uma integração do projecto desenvolvido no produto final da empresa.

3.2.1 Preparação em CPU

O servidor contém uma única *thread* em execução no CPU, esta tem a obrigação de controlar o uso do GPU com base nas tarefas que são submetidas a uma fila de tarefas presente no servidor. Esta fila é preenchida incrementalmente pelos processos de classificação enquanto estes fazem os seus pedidos, ou podem ser criadas pela *thread* principal do servidor, depois de ter dado como finalizada uma tarefa que se sabe ser a precedente à que vai ser criada.

Esta *thread* só termina a sua execução quando um sinal dado pela principal *thread* que atende pedidos recebidos de um cliente. Neste momento dá-se a libertação dos vários recursos tanto na memória do CPU como na memória do GPU.

3.2.1.1 Reserva de Recursos

Inicialmente o servidor começa por reservar vários buffers no GPU, quase esgotando toda a sua memória, mantendo alguma disponível para recursos que vão sendo alocados pela plataforma OpenCL durante a execução do programa. Posteriormente estes são divididos em fragmentos com base em valores estatísticos desta aplicação, tais como o número de n-gramas médio dos pedidos e *threshold* a partir do qual um processo deve ser processado em GPU.

Um conjunto de fragmentos de cada buffer servem de recurso para um pedido médio, no entanto existe uma maneira de atender pedidos que possam necessitar de mais recursos do que os alocados a um pedido com o tamanho usual. A estrutura que descreve um pedido, partilhada por GPU e CPU, tem a capacidade de repartir as suas estruturas por vários segmentos de buffers, tal como está representado na Figura 3.3. A reserva de mais fragmentos de buffer do que é usual é feita à conta de listas de apontadores, apontadores estes feitos à custa de índices visto ser este o suporte que o OpenCL permite. Na Figura 3.3 podemos ver esta técnica em relação às estruturas de dados utilizadas para

guardar uni-gramas, no entanto esta técnica é utilizada para todos os elementos enumerados na estrutura do pedido de maneira a conseguir lidar com vários tipos de variâncias no conteúdo e estrutura do documento.

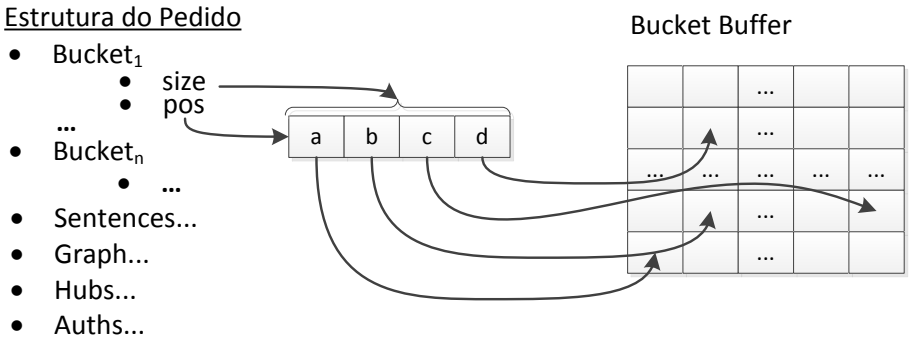

Figura 3.3: Estrutura dos meta-dados de um Pedido de Classificação de um Documento em CPU e GPU

Esta técnica tenta jogar com as probabilidades da existência de documentos de determinado tamanho no *corpus* a analisar. Normalmente as páginas web têm um tamanho médio que varia pouco, no entanto algumas podem ter um tamanho completamente desproporcional em relação a outras, no entanto este grupo é pequeno. Com este tipo de abordagem consegue-se aumentar em muito a capacidade de atender pedidos, enquanto não se perde a capacidade de atender pedidos com dimensões fora do comum.

Apesar de tudo, esta técnica pode ser definida como uma complexa e difícil gestão de memória e pode levantar alguns pequenos *overheads*, na ordem dos nano-segundos ou dos micro-segundos. Isto acontece porque nada garante que os fragmentos alocados são contíguos, o que pode levantar alguns *overheads* na leitura e escrita de memória visto que pode-se perder operações de leitura/escrita em zonas de memória contígua no GPU.

3.2.2 GPU *Multiple*SIMD (MSIMD)

Num servidor a quantidade de trabalho vai evidentemente aumentar, mas também a heterogeneidade de tarefas que podiam ser executadas simultaneamente. Utilizando o modelo do OpenCL não podemos fazer isto de maneira clara, mas podemos simular isto em GPU utilizando algumas abstracções. Estas abstracções são apresentadas de seguida e no fim serão apresentadas algumas optimizações que favorecem esta técnica em GPU.

Foi definido que uma tarefa podia ser dividida em várias partes, mas que um grupo de *threads* apenas poderia executar 1 destas partes e seriam sempre lançados em GPU um número de grupos igual à soma de partes de todas as tarefas a executar.

3.2.2.1 Definição de Tarefas

Uma tarefa é definida da mesma maneira tanto em CPU como em GPU, e segue a seguinte estrutura:

```
1 struct Task{
2     uint taskÍd;
3     uint requestÍd;
4     uint taskPart;
5     uint numberOfParts;
6 }
```

Os componentes que identificam esta tarefa definem como ela deve ser executada e tendo que dados como input e onde deve ser armazenado o output.

O primeiro componente, taskId, diz qual é a função que deve ser executada. Este identificador pode identificar uma tarefa única como uma normalização do grafo, ou uma fase de uma tarefa, por exemplo a difusão de pesos de um ciclo do algoritmo WHAKA.

O segundo componente tem como tarefa apontar a zona de memória onde estão os dados que definem o pedido que está a ser processado.

Por fim, os dois últimos componentes servem para que as *threads* presentes em GPU tenham noção do contexto que envolve aquela tarefa e possam dividir o trabalho sem duplicar resultados devido a más divisões do mesmo.

Por definição todos os grupos lançados pela plataforma OpenCL têm o mesmo tamanho, que nós definimos e mantemos durante toda a execução desta nossa abstracção do GPU, que vamos denominar de T. Também definimos que cada tarefa submetida é decomposta num determinado número inteiro de sub-tarefas. Para que cada *thread* possa calcular o seu id dentro de uma tarefa utiliza a expressão apresentada em Def. 2.

Definição 2 *Definimos $tkid$ como o identificar único de uma thread numa tarefa, definimos que o valor deste é: $tkid$ = get_local_id() + $taskPart * T$*

Quando uma tarefa é submetida no CPU à fila de tarefas esta é decomposta de maneira a que existam n sub-tarefas na fila sendo que cada uma corresponde a uma parte da tarefa inicialmente submetida, tal como é apresentado na Figura 3.4. Esta factor serve para ajudar a paralelizar as tarefas em GPU, visto que cada grupo lançado acede apenas a uma destas novas sub-tarefas e com base na informação contida na mesma situa-se no contexto da tarefa global onde esta está contida.

Se a atribuição de tarefas se baseasse numa simples cláusula *if-then-else* e utilizasse chamadas directas a funções poderia aumentar em demasia o uso de registos por parte de cada *thread*, diminuindo o número máximo de *threads* possivelmente activo em GPU. O próximo ponto aborda este problema e a solução encontrada para o mesmo.

Poupança de Registos Para maximizar o uso do GPU temos que manter activo o maior número de *threads* possível, logo temos que minimizar o número de registos utilizado por cada *thread*, para que este não se torne num factor limitador da ocupação do GPU.

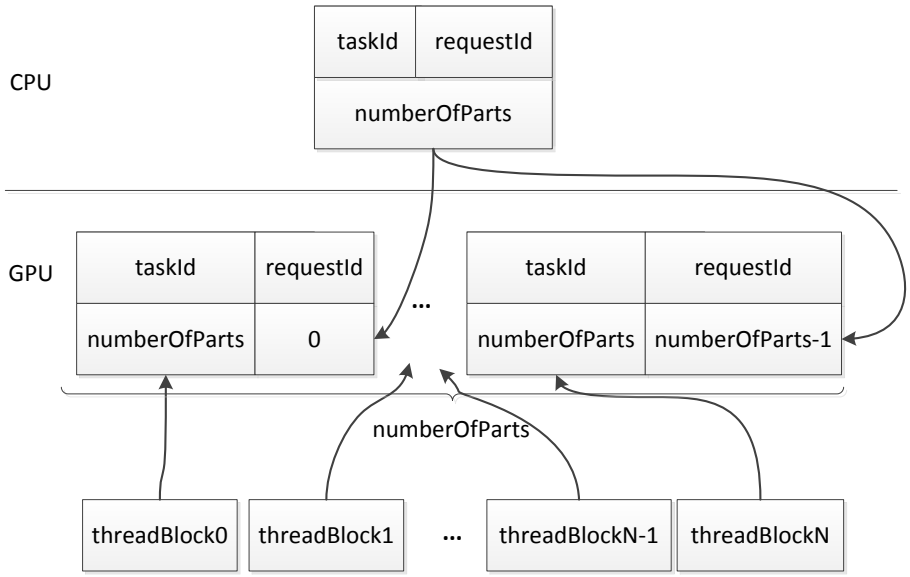

Figura 3.4: Decomposição de 1 tarefa em n sub-tarefas

O número de registos utilizados por cada é *thread* é calculado com base no código compilado. Para termos a certeza que esta reserva era minimizada, foi utilizado um vector para guardar os vários valores de cada *thread* e sendo reutilizado diferentemente por cada função criada.

A utilização do vector conseguia minimizar ligeiramente o uso de registos, mas dificulta a programação devido à sintaxe utilizada estar limitada. Para minimizar este problema, foi utilizada uma nomenclatura para substituir a nomeação do vector em várias funções com o auxílio da primitiva *#define* da linguagem.

3.3 Resultados e Análise

A análise à implementação concebida baseia-se em vários testes feitos sobre um conjunto de documentos, com tamanho variado de maneira a simular um fluxo de pedidos de classificação. Este conjunto é composto por um grande número de documentos com tamanho variado em que abundam os documentos com tamanho médio e sãos mais raros os que têm um tamanho pequeno ou grande, de maneira a tentar aproximar a simulação de um fluxo real de consulta de websites.

Os vários resultados apresentados são baseados em médias retirados de várias repetições dos mesmos testes. Esta opção tem como objectivo diminuir o ruído nos testes provenientes de factores exteriores ao mesmo. Não foram apresentadas barras de erro nos gráficos apresentados, porque o desvio nos valores dos vários testes em relação à média obtida é diminuto em relação às grandezas que são analisadas.

A versão em CPU irá utilizar cada um dos diferentes núcleos para um pedido de

classificação de forma independente. O protótipo em GPU irá fazer uso do módulo de gestão de processamento de pedidos por nós desenvolvido.

O principal interesse destes testes está no *throughput* que cada versão consegue obter em função de uma determinada velocidade de chegada dos pedidos. Apesar disto começaremos por fazer uma comparação e análise dos resultados obtidos entre esta implementação e a versão referida no capítulo anterior, onde o único interesse é o tempo de execução de um processo de classificação num ambiente fechado. Depois desta análise seguiremos então para análise dos resultados de *throughput* desta versão e da versão implementada em CPU.

Também será feito um esforço para deduzir quais as limitações de cada implementação. Por fim, conclui-se se ambas as implementações poderão trabalhar em conjunto de maneira a produzir um produto final de maior qualidade.

De seguida são apresentadas as configurações de hardware e software utilizadas nestes testes.

- **Contexto de Experimentação** As experiências foram feitas em 2 PCs distintos, que têm como hardware base: um processador Xeon E5504 (4-*core*); 12 Gbytes RAM; um GPU para a visualização e um acelerador.

  O que varia entre as duas máquinas é o acelerador que possuem, a máquina A tinha um nVidia C2050 (Fermi), que consiste num GPGPU com 448 núcleos CUDA, com 3 Gbytes de memória, A máquina B possuí um nVidia GTX680 (Kepler) com 1536 núcleos CUDA, com 2 Gbytes de memória.

  Relativamente ao software utilizado, este consistiu na plataforma OpenCL sobre o sistema operativo Linux na distribuição Ubuntu, versão 10.04.4, kernel 2.6.32-43; a versão do gcc usada é a 4.4. Convêm também referir que as máquinas utilizadas funcionam num ambiente partilhado entre múltiplos utilizadores, o que pode prejudicar ligeiramente o resultado de alguns testes, independentemente da versão a ser testada.

3.3.1 Processamento de uma Classificação(Algoritmo WHAKA)

Será aqui apresentada uma comparação entre as várias versões concebidas sobre este projecto. Como as alterações se focam todos na fase do algoritmo WHAKA, iremos apenas mostrar a relação que existe entre os vários resultados desta fase em relação às diferentes versões.

Os resultados que serão de seguida apresentados foram obtidos na máquina B, apresentada anteriormente que está equipada com um GPU nVidia GTX680.

Tal como pode ser visto na Figura 3.5 e na Figura 3.6 podemos identificar uma ligeira melhoria no desempenho da versão criada para processamento em *bulk* em relação à criada no capítulo anterior no processamento de um único pedido de classificação. Esta melhoria vai-se dissipando tornando-se mesmo numa ligeira perda de desempenho com o aumento do número de n-gramas de um documento.

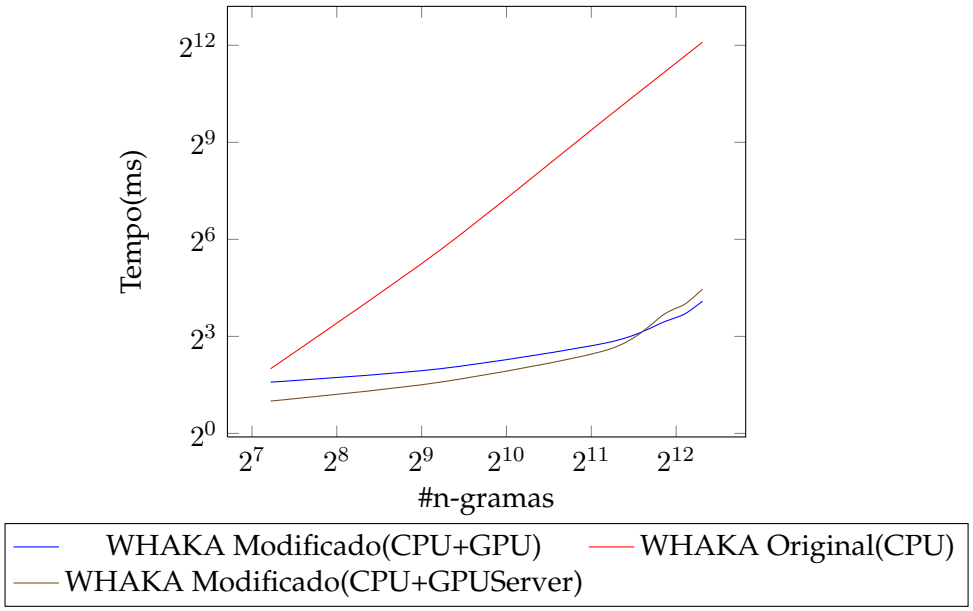

Figura 3.5: Tempos de Execução do Algoritmo WHAKA

A melhoria inicial é facilmente justificável, esta deriva da fusão de funções referida em 3.2.2.1. Esta técnica permite fundir funções que iriam ser executadas sequencialmente sem que provavelmente nenhuma fizesse um uso total dos recursos disponibilizados pelo acelerador. A fusão de uma ou mais funções pode aumentar ligeiramente o tempo de execução de uma função, mas irá quase sempre diminuir o tempo de execução do conjunto de funções em causa. Isto traduz-se de imediato num *speedup* do processo de classificação, visto que apesar do número de cálculos feito em GPU aumentar, este é escondido visto que o processo está limitado pelos acessos a memória e não por tempo gasto em cálculos.

Tal como seria expectável o tempo de execução desta versão começa a exceder o tempo de execução da versão anteriormente apresentada quando o número de n-gramas começa a exceder um determinado limiar. Este limiar está fortemente relacionado com os valores de parametrização do módulo. O factor que precisa de ser parametrizado é o número de n-gramas de um pedido dito de tamanho médio, este limite imposto artificialmente, obriga a que seja utilizada a técnica referida em 3.2.1.1 e que faz com que as várias *threads* não façam simplesmente leituras e escritas coalescentes, tornando o uso de memória mais demorado e aumentando o tempo total da classificação.

A perda de perfomance acentua-se quando desvantagem nas operações de leitura/escrita das várias *threads* se sobrepõem aos ganhos derivados da fusão de funções.

3.3.2 Processamento de um Conjunto de Pedidos

Esta sub-secção foca-se em fazer a recolha e análise de resultados da implementação criada. Inicialmente é feita uma análise da capacidade de atendimento do projecto em termos de documentos classificados por unidade de tempo e posteriormente é feita uma

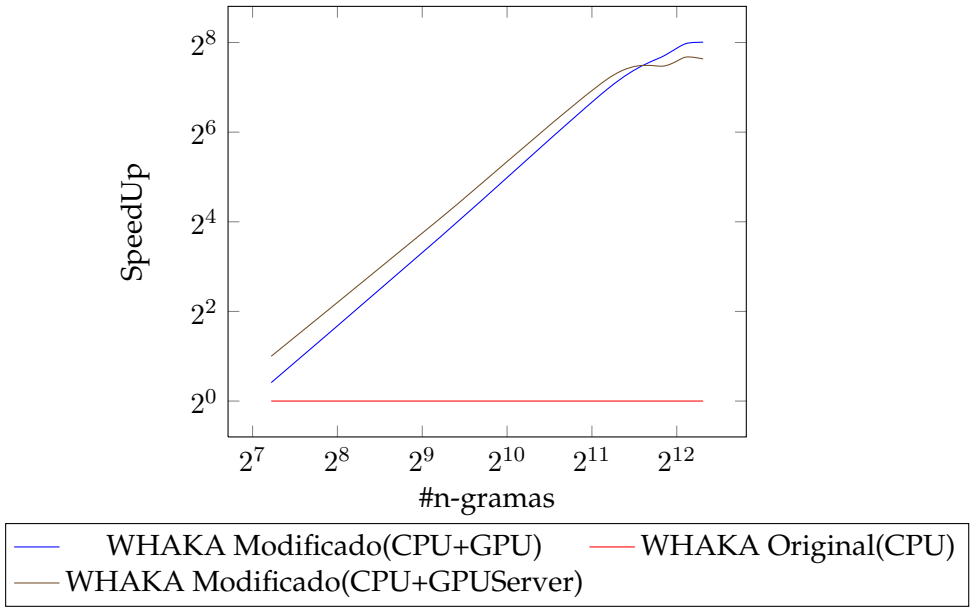

Figura 3.6: SpeedUp do Algoritmo WHAKA

análise para verificar o tempo médio que demora a classificação de um pedido.

O teste está limitado artificialmente pelo número de *threads* com permissão para executar no CPU, sendo este factor relevante em qualquer uma das implementações testadas. Este factor tem o seu limite superior definido para 64, o que serve perfeitamente para manter o CPU activo independentemente da versão do programa que está a ser utilizada, e o inferior para 5, visto que esta é uma limitação da biblioteca utilizada para auxílio na implementação, a biblioteca POCO[1]. Este número apenas reflecte o número máximo de pedidos a ser atendidos concorrentemente e não tem em conta as *threads* utilizadas na solução criada para execução de tarefas e libertação de recursos alocados por pedidos já atendidos.

A versão CPU apenas terá duas instâncias testadas, com 5 e 8 *threads*, visto que com apenas estas duas versões se consegue perceber a tendência das variáveis em estudo, percebendo-se que não existe a necessidade de testar de maneira mais exaustiva esta implementação. Em relação à versão GPU utiliza-se um mínimo de 8 *threads*, já que não existe qualquer interesse em analisar uma versão GPU capaz de atender os mesmos ou menos pedidos do que uma versão em CPU, visto que o objectivo desta fase do projecto é aumentar a capacidade de atender pedidos em simultâneo.

Para facilitar a análise dos dados considerou-se que um fluxo de pedidos chega a uma velocidade uniforme, ou seja, todos os pedidos de classificação estão espaçados entre si com um intervalo fixo de tempo. A velocidade deste fluxo de pedidos é um dos principais factores que vai ser tido em conta na projecção nos resultados apresentados.

[1]Biblioteca usada na versão original do projecto e contém primitivas para gestão de *threads*. A documentação está disponível em www.pocoproject.org

Quando não existem *threads* disponíveis para atendimento, os pedidos de classificação começam a ficar acumulados numa fila de pedidos pendentes para serem atendidos mal exista hipótese para que tal aconteça. Todos os pedidos estão definidos para serem executados em GPU, no entanto quando é identificada uma *thread* de atendimento que está livre e o GPU não tem definidos recursos para o atendimento do pedido, quando for atribuído um pedido de classificação à *thread* poderá manter o tratamento do pedido no CPU. Este é um comportamento que será mais usual quando a aplicação for submetida a uma grande carga, que poderá ser representada com base nos tamanhos dos pedidos ou na velocidade com que estes chegam.

3.3.2.1 Descrição dos Testes

Foram desenvolvidos 4 conjuntos de documentos, estando cada um associado a um único teste, ou seja, a um fluxo de pedidos. Os vários testes tem diferentes composições de documentos para tentar demonstrar os resultados do projecto quando na presença de diferentes factores.

Os testes e as suas composições são apresentados de seguida:

- **A) Pequena Escala**- Este teste é composto por 2484 réplicas de um documento com um número reduzido de n-gramas, neste caso com 121 n-gramas. Quando um pedido destes é atendido isoladamente em CPU demora cerca de 5 ms a ser atendido, com o auxílio do GPU demora 12 ms a ser atendido.

- **B) Média Escala**- Este teste é composto por 2401 réplicas de um documento com tamanho médio em termos de n-gramas, neste caso com 389 n-gramas. Quando um pedido destes é atendido isoladamente em CPU demora cerca de 27 ms a ser atendido, com o auxílio do GPU demora 26 ms a ser atendido.

- **C) Grande Escala**- Este teste é composto por 2424 réplicas de um documento com um número elevado de n-gramas, neste caso com 2180 n-gramas. Quando um pedido destes é atendido isoladamente em CPU demora cerca de 799 ms a ser atendido, com o auxílio do GPU demora 155 ms a ser atendido. É importante notar que este documento tem aproximadamente metade do número de n-gramas do maior documento utilizado nos testes do Capítulo 2, mas é muito mais comum na web do que esse.

- **D) Conjunto Misto**- Este teste é composto por 2485 documentos com tamanho variado em termos de n-gramas. Esta variação serve para tentar simular um fluxo real de pedidos de classificação, por essa mesma razão existem pedidos de tamanho médio em maioria e uma escassez de pedidos com um número elevado ou reduzido de n-gramas.

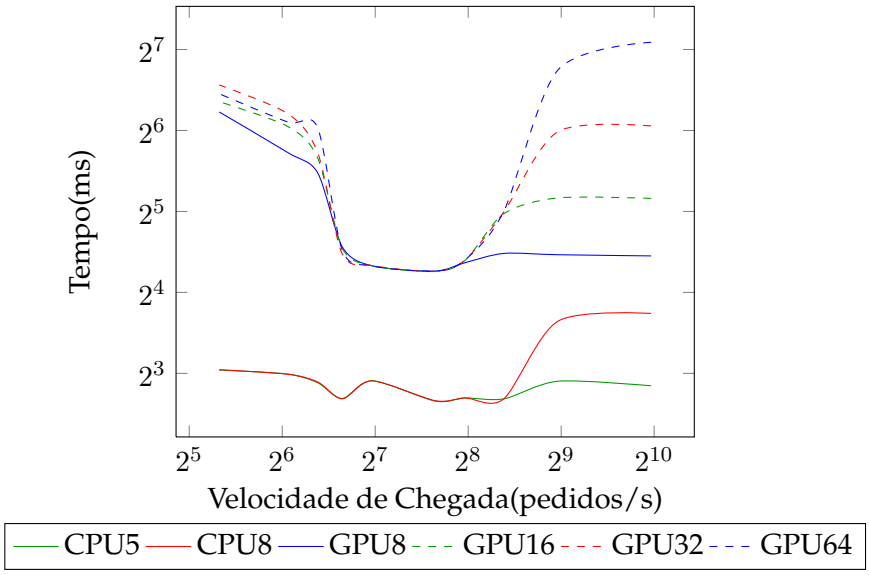

Figura 3.7: Tempos Médios de Processamento de um Pedido para o Teste A

3.3.2.2 Resultados

Em todas as figuras que serão apresentadas estão representadas várias séries de dados, sendo que cada uma diz respeito a uma implementação, CPU ou GPU, e o correspondente número de *threads* disponível em CPU para atendimento de pedidos de classificação. A nomenclatura é apresentada no formato HT onde H pode ser CPU ou GPU, conforme a correspondente versão, e T é o número de *threads* disponibilizadas para atendimento.

Os resultados que serão de seguida apresentados foram obtidos na máquina B, apresentada no início do capítulo com um GPU nVidia Tesla C2050. Apenas o último teste tem como contexto de experimentação os dois apresentados no início do capítulo visto que o objectivo deste teste é comparar esses mesmos contextos.

Teste de Pequena Escala(A) Para o tipo de documento presente neste teste a versão CPU deste projecto tem melhores resultados ao fazer apenas uma classificação. É expectável que tal continue a acontecer com um fluxo de pedidos de classificação composto apenas por pedidos deste tipo. Logo o nosso objectivo é ver como o *throughput* das versões CPU e GPU variam quando submetidas a diferentes cargas e quando é que a vantagem que o CPU tem sobre GPU se dissipa devido à acumulação de pedidos de classificação na fila de espera.

São apresentadas 3 figuras para demonstrar os resultados obtidos, sendo que uma apresenta o tempo médio de processamento de um pedido, Figura 3.7, uma apresenta o tempo total de espera pela resposta de um pedido, Figura 3.8, ou seja, o tempo de processamento mais o tempo de espera. Por último é apresentado na Figura 3.9 o *throughput* que cada tipo de implementação tem e como este varia em função da velocidade de chegada de pedidos.

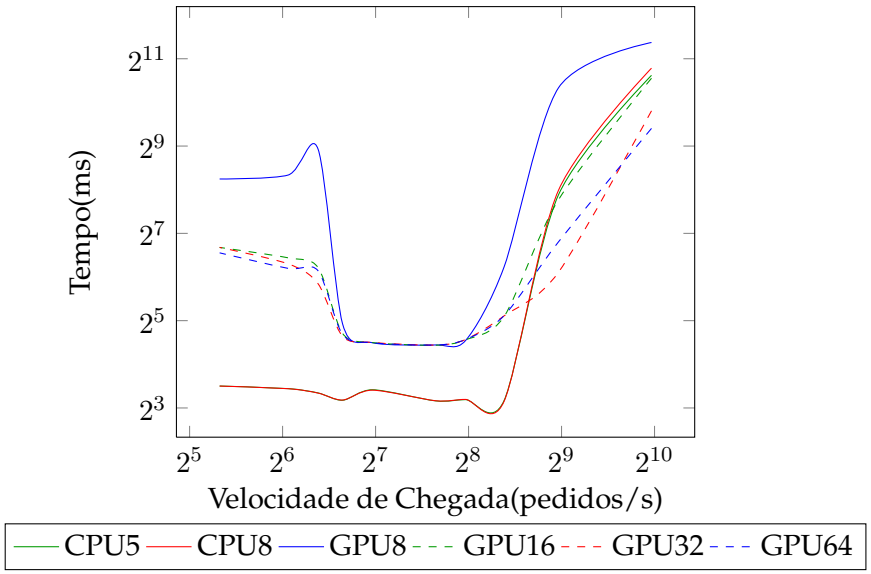

Figura 3.8: Tempos Médios de Resposta de um Pedido para o Teste A

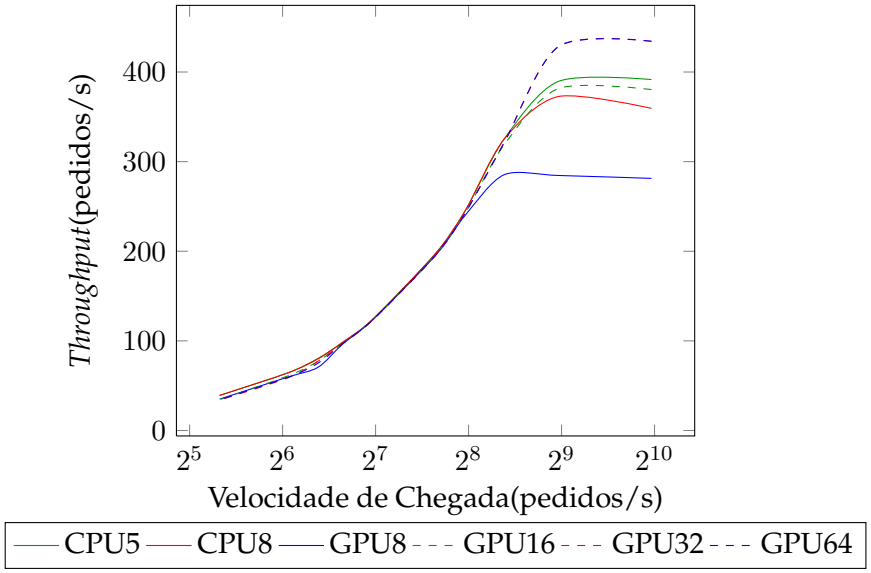

Figura 3.9: *Throughput* para o teste A

Podemos ver que o aumento do número de *threads* de atendimento na versão CPU não traz grandes vantagens a este implementação. É visível que a versão CPU que melhores resultados traz é a representada pela série "CPU5", conseguindo melhores tempos médios de atendimento e *throughput*. A versão "CPU8"começa a perder performance devido ao custo de gestão e execução de mais *threads* do que a primeira versão referida. Tendo isto em mente tornaremos a versão CPU5 como a comparada por omissão com as versões GPU testadas.

Em relação ao tempo médio de processamento, podemos ver que este não varia muito, com o CPU a conseguir quase sempre atender os pedidos em menos de 10 ms, embora perca em demasia no tempo de espera dos pedidos, visto que o número de pedidos tratados por unidade de tempo é inferior ao número de pedidos que chegam no mesmo período.

Tanto o tempo médio de processamento como o tempo médio de resposta contêm uma anomalia para as versões GPU testadas para velocidades de chegada de pedidos inferiores a 128 pedidos/s. Esta anomalia é considerada desprezável visto que pode ser corrigida com uma parametrização cuidada da versão testada, tal como é apresentado no último teste deste capítulo. Em última instância pode-se sempre afirmar que se conseguem obter resultados tão bons como os obtidos para velocidades de chegada iguais a 128 pedidos/s, nem que seja com recurso a um aumento de carga simulado.

O GPU independentemente da velocidade de chegada de pedidos e do número de *threads* de atendimento irá manter um tempo médio de processamento superior à versão CPU, no entanto podemos obter melhores *throughputs* nesta implementação, devido à acumulação de pedidos na versão CPU que deriva de uma carga demasiado elevada sobre o servidor. Esta acumulação em nada afecta o tempo de processamento em CPU, mas tem um efeito extremamente negativo sobre o tempo de resposta do mesmo, visto que um pedido pode ter de ficar na fila de espera durante um largo intervalo de tempo à espera de ser processado. Para GPU este cenário não é aplicável, como se tenta maximizar o número de pedidos a serem atendidos concorrentemente, conseguimos diminuir este tempo de espera e de imediato diminuir o tempo médio de resposta do servidor.

Podemos concluir que para pedidos com um tamanho diminuto, o GPU consegue ter ganhos em cenários onde a carga sobre o servidor é relativamente elevada, beneficiando o sistema de um ponto de vista global, conseguindo responder aos pedidos mais depressa do que a versão em CPU. No entanto este cenário só se torna interessante e mesmo aplicável quando o número de *threads* de atendimento se torna razoável, 32 ou 64, e quando a carga sobre o servidor é maior ou igual a 400 pedidos/s, sendo a versão com 16 *threads* de atendimento quase equivalente em termos de *throughput* à versão implementada em CPU.

Neste termos podemos declarar como proveitoso uma implementação capaz de fazer *offloading* de trabalho de CPU para GPU quando este começa a ficar com uma fila de espera cheia.

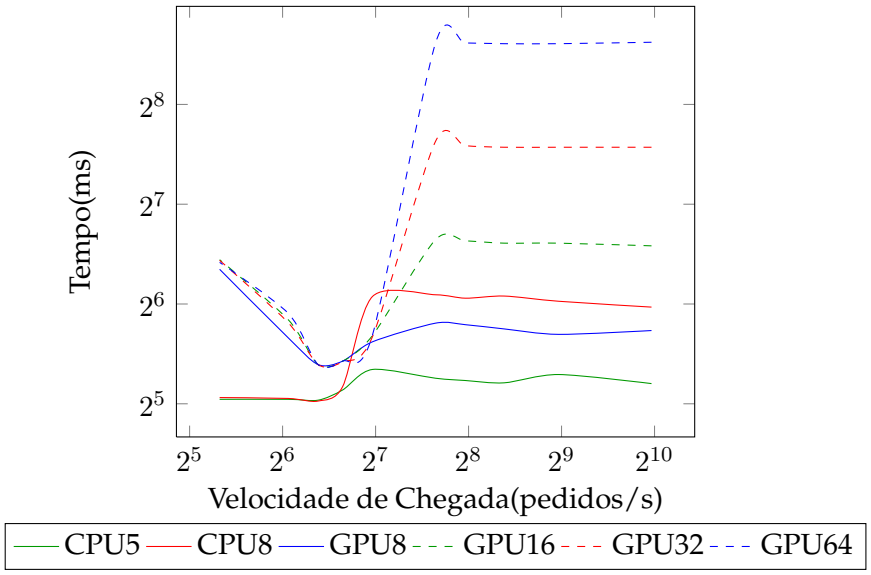

Figura 3.10: Tempos Médios de Processamento de um Pedido para o Teste B

Teste de Média Escala(B) Neste teste tentou-se escolher um documento que tivesse um tempo de processamento similar entre CPU e GPU. Podemos então considerar este teste o mais imparcial e justo entre as duas versões. Assim espera-se conseguir ver como variam os resultados das duas versões quando submetidas a diferentes cargas, quando estão presentes valores que pouco variam em termos de tempo de execução quando são processados num ambiente fechado.

Tal como no teste anterior continuamos a ter resultados com a sua informação repartida por 3 gráficos. Estes contêm o mesmo tipo de informação, tempos médios de processamento - Figura 3.10, tempos médios de resposta a um pedido - Figura 3.11 e por último o *throughput* - Figura 3.12 - deste teste.

Segundo o gráfico presente na Figura 3.10 podemos observar a versão CPU com 5 *threads* tem sempre tempos de processamento inferiores aos de qualquer versão em GPU, tal também é quase sempre uma realidade para a versão CPU com 8 *threads*. Ou seja, a ligeira vantagem que o GPU tem em relação á versão do CPU em relação à classificação de pedidos num ambiente fechado em que apenas um pedido é classificado de cada vez dissipa-se neste contexto.

No entanto, este factor torna-se algo irrelevante quando a carga enviada para o servidor se aproxima dos 100 pedidos/s. Neste momento, apesar de um tempo de processamento menor por parte do CPU, podemos observar na Figura 3.11 que o tempo que demora a resposta a um pedido se torna superior para a implementação em CPU em relação à versão em GPU. Isto deve-se à técnica de *bulk/batch* desenvolvida na solução criada, que consegue aumentar o *throughput* em detrimento do tempo médio de processamento, isto leva a uma diminuição do tempo de espera por parte dos pedidos na versão GPU.

As considerações a fazer sobre este teste são as mesmas que foram referidas em 3.3.2.2, apenas com a diferença de que o limite é atingido com um ritmo de chegada de pedidos

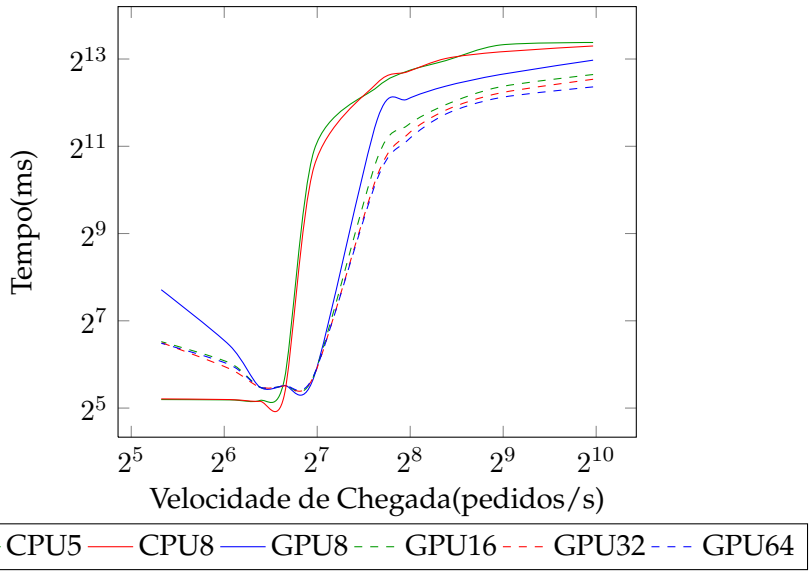

Figura 3.11: Tempos Médios de Resposta de um Pedido para o Teste B

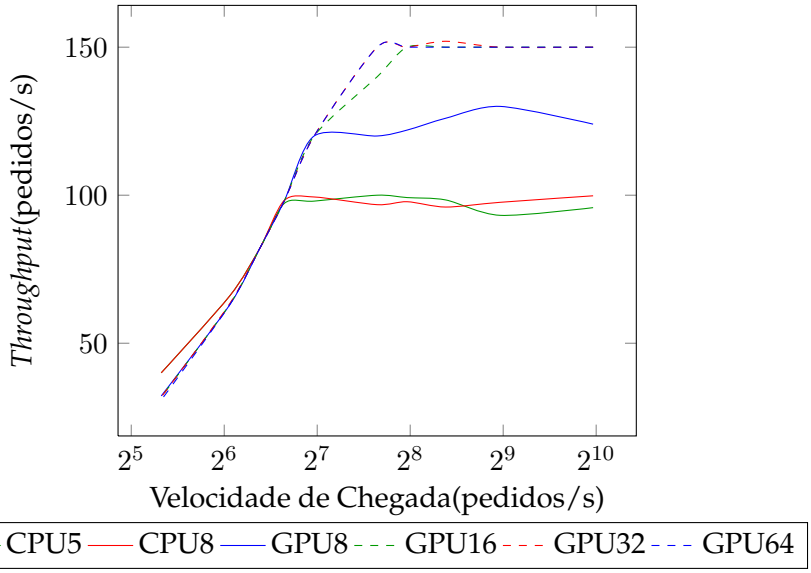

Figura 3.12: *Throughput* para o teste B

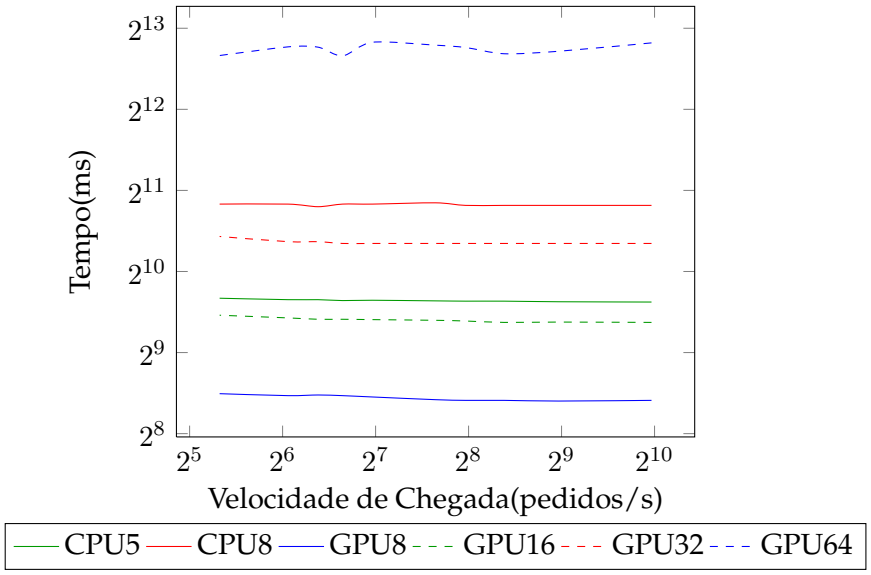

Figura 3.13: Tempos Médios de Processamento de um Pedido para o Teste C

de 100 por segundo.

Podemos identificar uma barreira ao 100 pedidos/s na velocidade de chegada de pedidos para a versão CPU. É nas imediações deste ponto que a versão CPU começa a perder a sua capacidade de aumentar o *throughput* do servidor, mantendo-se este num valor aproximado dos 100 pedidos/s daí para a frente. Em relação ao *throughput*, podemos observar que a versão GPU do projecto está menos limitada, conseguido obter no limite um *throughput* de 150 pedidos/s e conseguindo manter um tempo médio de resposta sempre que chega a este tipo de valores.

Teste de Grande Escala(C) Este teste é composto pela classificação de várias cópias de um documento com um tamanho relativamente grande, em termos de número de n-gramas. Supostamente este seria o caso em que o GPU seria mais beneficiado, visto que já na classificação de apenas uma instância deste documento já se notam enormes ganhos entre GPU e CPU. O principal objectivo deste teste é analisar o comportamento das várias versões do projecto quando são submetidas várias cargas, quando os pedidos feitos já exigem o uso de bastantes recursos, traduzindo-se em tempo para CPU e memória para GPU.

Na Figura 3.15 podemos observar que o *throughput* da implementação em GPU continua a ser superior em relação à implementação dependente unicamente do CPU. Este cenário dá-se independentemente do número de *threads* de atendimento, visto que o tamanho dos pedidos são de uma escala superior ao usual, sendo sempre feito um uso de todos os recursos do GPU, independentemente se existem 8, 16, 32 ou 64 *threads* de atendimento.

Nos testes anteriores podemos observar que se maximiza o *throughput* à custa do aumento dos tempos de processamento de alguns pedidos.

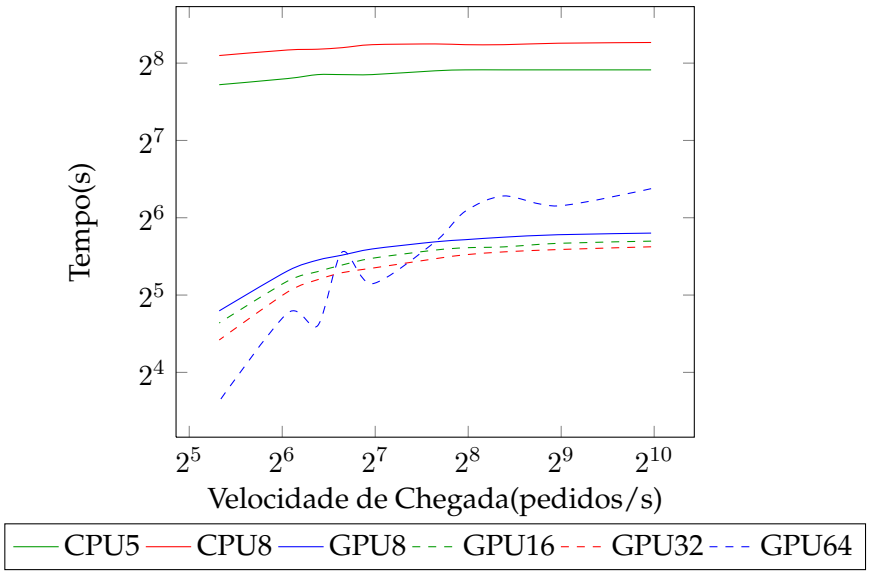

Figura 3.14: Tempos Médios de Resposta de um Pedido para o Teste C

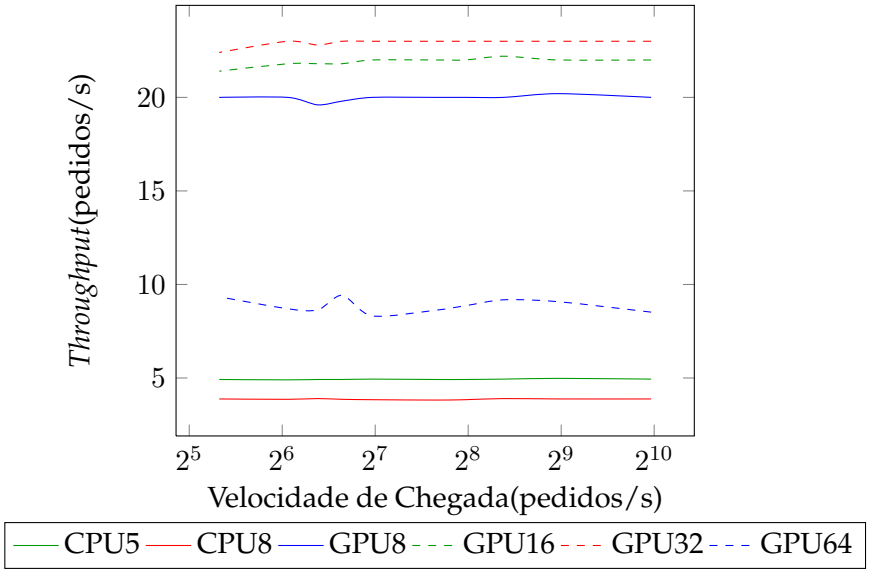

Figura 3.15: *Throughput* para o teste C

Podemos observar os tempos de processamento na Figura 3.13 e com uma análise bastante simples concluímos que a versão GPU do projecto tem muitas vezes tempos de processamento menores do que a versão em CPU. Isto deve-se à magnitude do pedido, como o número de n-gramas é elevado e existe um aumento substancial do esforço em CPU para computar o resultado do algoritmo estudado, enquanto o GPU ganha uma melhor ocupação dos seus recursos.

Os recursos *hardware* do GPU são limitados, por isso apenas uma pequena percentagem dos pedidos enviados entram de imediato em execução, sendo uma parte destes enviados para CPU quando existe uma *thread* de atendimento disponível. O aumento do número de *threads* de atendimento pode piorar o tempo médio de execução, já que isto pode obrigar a que mais pedidos sejam enviados do GPU para CPU.

A técnica de *bulk/batch* tem as suas vantagens representadas entre a Figura 3.14 e Figura 3.13. A diferença que existe entre os tempos dos dois gráficos consegue o tempo médio que um pedido passa na fila de espera, conclui-se que este tempo também é menor para GPU do que para CPU.

O uso de 64 *threads* devolve resultados extremamente irregulares, isto deve-se à falta de capacidade de executar o grande número de tarefas que retorna que vêm de todos os pedidos e da capacidade de manter a sua informação em memória. Neste caso e devido às características do fluxo de pedidos existem vários casos em que o tratamento do pedido se mantém em CPU. Este contexto torna irregulares os resultados obtidos para um número tão alto de *threads* de atendimento. Isto não se verifica com quantidades menores de *threads* de atendimento, visto que o menor número tende a linearizar os valores dos resultados obtidos, devido a pedidos que são *offloaded* para CPU.

Teste Misto(D) Este teste tem como objectivo aproximar-se de um contexto real de uso do classificador, apesar de manter alguns factores controlados de maneira a facilitar a análise de resultados. Num contexto real seria de esperar uma velocidade de chegada de pedidos variável e possivelmente até bastante irregular. Este seria um cenário interessante, mas além de haver alguma dificuldade em reproduzir um réplica de um fluxo de pedidos real também iria existir uma grande agravante na análise dos resultados obtidos devido à aleatoriedade que poderia vir do padrão de chegada destes mesmo pedidos.

Tentou-se então usar este contexto para simular um cenário que estivesse o mais aproximado da realidade quanto fosse possível. Assim poderia analisar o comportamento da versão CPU e GPU num cenário aproximado da realidade.

São apresentados 2 figuras para demonstrar os resultados obtidos, sendo que uma apresenta o *throughput* de cada implementação, Figura 3.17, tal como foi sempre apresentado nos testes anteriores.

A segunda figura, Figura 3.16, tem como objectivo mostrar a eficiência relativa de cada implementação em relação ao seu objectivo final, classificação rápida e no maior número de pedidos possível. Ou seja, foi definido um *threshold* para o tempo efectivo de resposta a um pedido, o valor definido foi de 1 segundo.

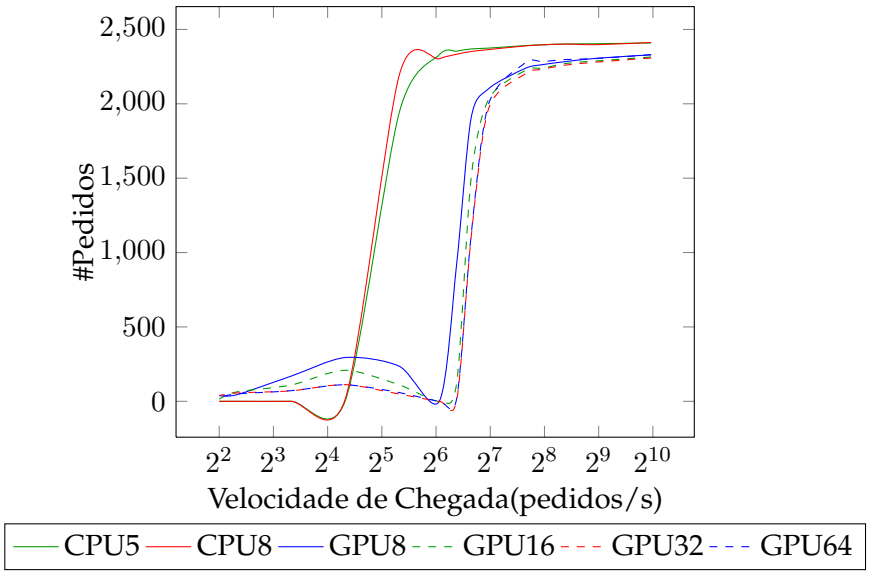

Figura 3.16: Documentos que excedem o tempo razoável de classificação

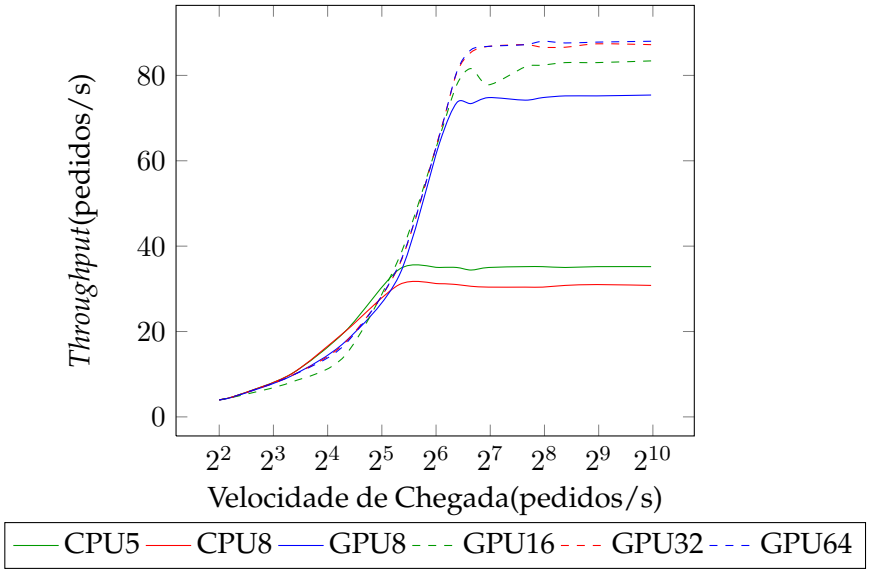

Figura 3.17: *Throughput* para o teste D

O melhor resultado seria da implementação que conseguisse o maior *throughput* e que conseguisse responder em tempo útil a todos os pedidos.

Segundo o gráfico representado na Figura 3.17, podemos observar que o *throughput* de qualquer instância da versão implementada em GPU tem acesso a valores melhores quando a velocidade de chegada de pedidos começa a ser maior do que 40 pedidos/s. Este é um resultado extremamente positivo, visto que se espera que o classificador seja sujeito a uma carga superior a esta quando estiver num contexto real de utilização.

O *throughput* pode ser uma medida algo enganadora e apesar do mesmo ter bons valores, não quer dizer que os pedidos sejam atendidos num intervalo de tempo razoável. Analisando a informação presente na Figura 3.16 podemos ver que para cargas grandes, sensivelmente acima de 90 pedidos/s, qualquer uma das implementações começa a não conseguir atender quase nenhum pedido em tempo útil. No entanto, devido à técnica de *bulk/batch* utilizada na versão GPU do projecto é notável uma diferença substancial nos valores de classificações em tempo útil para velocidades de chegada menores do que 90 pedidos/s.

Enquanto isto a versão em CPU continua a perder muitos pedidos para velocidades de chegada menores do que 90 pedidos/s, apenas começando a recuperar de maneira aceitável quando a velocidade de chegada é igual ou menor a 20 pedidos/s. A variação destes valores em CPU é muito menos violenta do que em GPU, visto que não existe qualquer tipo de dependência entre os vários pedidos. Neste contexto apenas os pedidos que começam a ser atendidos mais tarde é que passam o *threshold*, visto que estiveram bastante tempo na fila de espera.

Pode-se concluir que a implementação em GPU é melhor do que a que apenas utiliza CPU para cargas superiores a cerca de 90 pedidos/s, valor este que se espera estar abaixo da velocidade de chegada de pedidos num contexto real de utilização do classificador.

Tal como nos testes até agora feitos, consegue-se ver algumas anomalias. Estas anomalias acontecem para as implementações em GPU quando submetidas a testes que têm uma velocidade de chegada de pedidos superior a 64 pedidos/s. No entanto é razoável inferir que para velocidade de chegada menores do que esta se continuam a conseguir atender todos os pedidos abaixo do *threshold* definido acima, com base numa parametrização do projecto. No último teste é apresentado um cenário semelhante ao apresentado neste teste onde se pode verificar que esta inferência é justificada.

Teste ao *Hardware* Durante toda a dissertação foram mostrados testes em duas máquinas, onde apenas variava o GPU presente em cada uma, um nVidia Tesla C2050 e um nVidia GTX680.

Este teste tem como objectivo verificar o comportamento do projecto final desenvolvido, quando executado em dois dispositivos de *hardware* diferentes

Tendo em vista o objectivo final do projecto e os resultados até agora apresentados decidiu-se executar a aplicação sobre o conjunto misto de documentos. Com base nos resultados anteriormente apresentados podemos ver que alguns do melhores resultados

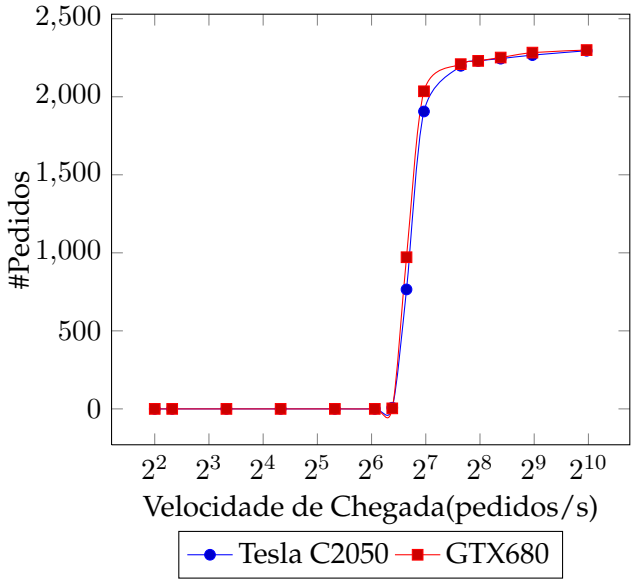

Figura 3.18: Documentos que excedem o tempo razoável de classificação em 2 GPU

advém da definição de 32 *threads* para atendimento de pedidos, logo esta será a configuração utilizada para comparação entre os dois contextos.

Independentemente do resultados que serão apresentados, será sempre correcto afirmar que em termos de capacidade de atendimento de pedidos em simultâneo num Tesla C2050 é superior à de um GTX680, devido à diferença de memória que existe entre os dois dispositivos. Contudo, visto que também não poderemos aumentar o número de pedidos atendidos em simultâneo em demasia, é razoável considerar que a capacidade de memória menor, 2 GigaBytes, é a suficiente para tratar dos problemas até agora apresentados.

Os dados aqui apresentados não sofrem dos *overheads* apresentados nos testes anteriores que se situavam entres as velocidade de entrada 4 e os 128 pedidos/s. A ausência desta anomalia deve-se a ligeiras diferenças feitas na configuração da instância da aplicação testada. Esta instância teve um corte no tamanho da memória alocada em GPU, para conseguir ter uma base de comparação similar entre os dois GPU testados. Estes dados justificam a afirmação feita para o teste A e para o teste B de que as anomalias lá identificadas podem ser ignoradas, uma vez que é sempre possível garantir um ritmo mínimo de pedidos.

Em relação aos resultados comparativos do *hardware* vemos que os diferentes GPU têm resultados que são muito semelhantes. Este é um factor de alta relevância para a produção em massa deste projecto, devido principalmente aos custos de *hardware*.

Enquanto um GPU cientifico de alta performance, embora já um pouco antigo - nVidia Tesla C2050, custa cerca de 2000 dólares actualmente, um GPU como o nVidia GTX680 custa cerca de $\frac{1}{4}$ deste valor. Esta conjugação de uma negligenciável variação de performance com uma diferença de preço considerável leva à conclusão de que a produção em

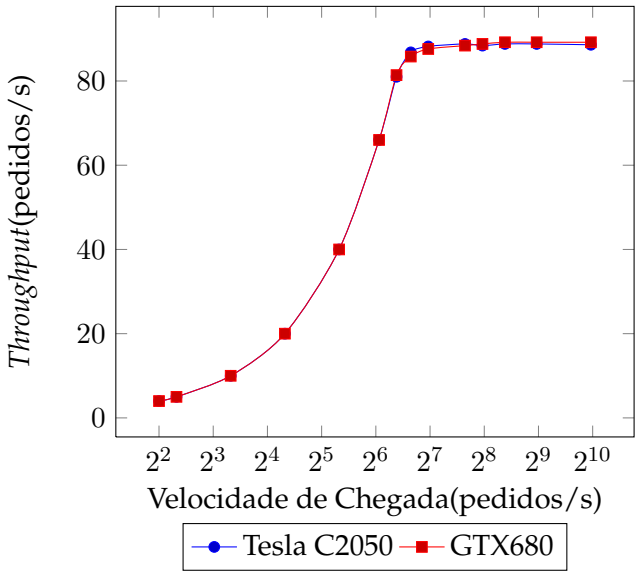

Figura 3.19: *Throughput* para o teste D para 2 GPU

massa do produto tem como opção viável e muito atractiva a utilização de GPU reconhecidos como GPU criados para jogos.

3.3.2.3 Análise Geral

Com base no resultado de todos os testes apresentados, pode-se concluir que a solução criada atinge os objectivos pretendidos, especialmente para problemas que sejam baseados na classificação de documentos compostos por mais n-gramas do que é usual. Esta característica deve-se especialmente ao uso proveitoso dos recursos do GPU na sua totalidade. Para problemas com tamanhos relativamente pequenos podemos concluir que o CPU continua a ser um bom candidato para o processamento dos mesmos, embora seja bem mais difícil sobrecarregar o GPU com uma grande carga de pedidos do que o CPU. Ou seja, o GPU consegue ser melhor na classificação de pequenos documentos, quando bastantes documentos deste tipo chegam ao servidor para classificação num pequeno intervalo de tempo.

Os casos apresentados são os dois extremos do problema apresentado, no entanto o mais comum dos casos é termos um pedido de classificação com um tamanho médio em termos de n-gramas, digamos entre 350 e 850 n-gramas. Este tipo de problema tem tempos e execução ligeiramente inferiores em GPU, se tomarmos em conta o tempo de espera previsto com base numa velocidade de chegada de pedidos razoável, e a capacidade de *throughput* também é maior neste tipo de hardware. Isto é um forte incentivo em relação ao uso de CPU auxiliado por GPU, ao invés de apenas utilizar um deles.

O teste mais interessante e próximo do real é o teste D. Este teste mostra que a implementação em GPU supera a implementação apenas baseada em CPU para cargas de trabalho irregulares e portanto mais similares a fluxos de pedidos de classificação reais.

Conclui-se então que a solução implementada não deve substituir a já existente na sua

totalidade, mas deve sim auxiliar a mesma de maneira a criar valor acrescentado em termos de desempenho global. Utilizando CPU e GPU como dois dispositivos de hardware com preferência de processamento por pedidos que se encaixem em determinado perfil, auxiliando-se mutuamente quando submetidas a situações de carga de grande escala. Esta conclusão torna-se ainda mais forte quando analisamos os resultados de processamento em GPU quando submetido a cargas que obrigam ao seu uso total deixando o CPU desocupado. Pode-se até conjecturar que o melhor desempenho que o GPU terá será em ambientes irregulares em termo de fluxo de pedidos e do tamanho dos mesmos, sendo este o contexto real a que a aplicação deverá responder.

A conjugação das duas soluções deverá resultar num classificador capaz de classificar documentos de pequeno porte em CPU de maneira rápida, fazendo *offload* destes para GPU quando é identificada uma carga capaz de aumentar a fila de pedidos em demasia. Esta também deve conseguir utilizar heurísticas de maneira a saber se será mais benéfico processar um pedido de classificação em CPU ou GPU, com base no tamanho do pedido, nos recursos de hardware presentes na máquina e nos recursos disponíveis no momento de recepção desse mesmo pedido.

O produto final deverá ser de um custo relativamente reduzido devido aos preços actuais do tipo de hardware em uso. Também é expectável o desenvolvimento de um produto final de grande capacidade devido à conjugação de diversas arquitecturas hardware, especificamente CPU e GPU.

4

Conclusão e Trabalho Futuro

Este capítulo tem como objectivo apresentar o balanço geral da dissertação desenvolvida e apresentar algumas das opções de trabalho futuro.

4.1 Conclusões

Através do projecto realizado como objectivo da presente dissertação, foi possível criar um algoritmo sobre GPU que permite ajudar no cálculo da relevância de n-gramas em vários documentos concorrentemente. Esta implementação vai permitir distinguir os n-gramas mais importantes num documento e logo as *features* que podem ajudar o mesmo a ser classificado.

Em relação ao processamento de documentos de grande complexidade conseguiu-se criar uma versão que executa em GPU, que tem tempos de execução 16 vezes inferiores aos conseguidos pela versão em CPU. Estes ganhos não são tão expressivos para documentos mais simples.

Este projecto irá permitir à empresa desenvolver um produto com maior capacidade e velocidade de atendimento do que um baseado apenas em CPU, por um custo que se pode tornar significativamente menor que a produção do projecto em massa recorrendo apenas a CPU, devido ao custo de componentes e ao consumo de energia menor dos mesmos.

Convém referir que a solução concebida é compatível com uma grande variedade de aceleradores, no entanto pode precisar de mais algum trabalho sobre a alocação de recursos no acelerador em uso.

A solução desenvolvida consegue fazer uso de vários aceleradores simultaneamente,

no entanto para que esta opção se torne apetecível de utilizar terá que se criar um distribuidor de carga, capaz de lidar com a heterogeneidade dos processos de classificação e dos vários aceleradores.

Existem diversos parâmetros que podem e devem ser alterados quando o projecto está a ser instalado numa máquina, estes são parâmetros que ajudam na distribuição de trabalho entre CPU e GPU e na alocação de memória do GPU. Vários outros parâmetros derivam do algoritmo original que foi transformado para ser executado em GPU, estes são tratados de maneira semelhante à maneira como eram tratados na versão anterior do projecto e podem ser configurados da mesma maneira. No entanto a modificação de algumas regras matemáticas do algoritmo podem se traduzir em mudanças mais complexas no código criado para ser executado em GPU, sendo este um dos maiores incómodos da solução criada.

Pode-se analisar os detalhes da programação em GPGPU, tornando possível detectar as principais limitações deste dispositivos e elevadas capacidades computacionais dos mesmos. As principais vantagens ficam focadas na grande capacidade destes dispositivos, enquanto como principal desvantagem podemos indicar a dificuldade de programação e principalmente de *debug* neste tipo de ambiente. O uso da plataforma OpenCL traduz-se na vantagem da portabilidade do código produzido, embora a portabilidade não garanta o uso óptimo de recursos em diferentes tipos de hardware.

4.2 Trabalho Futuro

Depois da revisão da solução implementada foram identificados alguns temas que podem ser trabalhados futuramente de maneira a melhorar o desempenho do projecto desenvolvido.

Seria de extrema importância construir um distribuidor de carga, de maneira a não prejudicar em demasia o tempo de processamento de qualquer pedido de de classificação e dando a capacidade de gerir de forma mais eficaz os vários recursos do sistema, CPU e um ou mais GPGPU.

Um outro tema interessante seria definir uma técnica para lidar com as cargas de trabalho extremamente irregulares que se encontram ao longo das várias operações no processo analisado e modificado. As tarefas onde esta técnica poderia ser mais benéfica seriam as que são marcadas por uma grande irregularidade nos diversos trabalhos que geram para as *threads* que os vão executar. Uma possível solução poderia estar na definição dinâmica de grupos de *threads* capazes de calcular o tamanho do grupo com base no tamanho da tarefa a processar, conseguindo controlar o uso de memória local e global utilizada em todo o processo.

Apesar de este não ser um limitador da implementação, seria positivo criar uma técnica de alocação de memória em função das várias características do dispositivo, GPGPU,

a ser utilizado. Esta possível tarefa futura entra em harmonia com o tema de maior importância que pode ser trabalhado tendo esta tese como base. Existe uma grande necessidade de abstracção do GPGPU, devido à sua dificuldade de programação, optimização e utilização.

Existem imensas tarefas que podem ser beneficiadas do uso deste tipo de hardware, no entanto estas não são complexas o suficiente para fazerem um uso completo dos recursos disponíveis. Para maximizar o uso deste hardware, sem abdicar de alguma abstracção, seria de grande interesse e importância criar uma plataforma com um comportamento semelhante ao criado no Capítulo 3. Esta plataforma deveria ter primitivas para adição de tarefas genéricas ao programa executado em GPU, primitivas de alocação de memória e por fim primitivas de I/O capazes de trabalhar de maneira transparente para o utilizador da plataforma. As primitivas desta plataforma deveriam ser na sua essência muito semelhantes às apresentadas em 3.2 para a abstracção GPUServer, definida explicitamente para este projecto. A ideia final desta plataforma seria trabalhar como um servidor disponível para uso por parte de vários processos presentes em CPU, de maneira a transportar o trabalho destes processos para GPU e esconder toda a complexidade de gestão de memória e do fluxo de execução do hardware.

Prevê-se que estes desenvolvimentos sejam feitos pela empresa, uma vez que o trabalho conducente a esta dissertação demonstrou a viabilidade técnica e económica da abordagem de utilizar GPUs para a aceleração do processamento de documentos.

Bibliografia

[CAP11] nVidia The Portland Group CAPS, Cray Inc. OpenACC, 2011.

[CB99] E. Christen e K. Bakalar. Vhdl-ams-a hardware description language for analog and mixed-signal applications. *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on*, 46(10):1263–1272, 1999.

[Cor12] nVidia Corporation. Kepler GK110 architecture, 2012.

[DHH+02] C. Ding, X. He, P. Husbands, H. Zha, e H.D. Simon. Pagerank, HITS and a unified framework for link analysis. In *Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval*, pág. 353–354. ACM, 2002.

[DLN+94] J. Dongarra, A. Lumsdaine, X. Niu, R. Pozo, e K. Remington. A sparse matrix library in c++ for high performance architectures. 1994.

[DM98] L. Dagum e R. Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[Fly72] Michael J. Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, C-21(9):948 –960, sept. 1972.

[FVS11] J. Fang, A.L. Varbanescu, e H. Sips. A comprehensive performance comparison of CUDA and OpenCL. In *Parallel Processing (ICPP), 2011 International Conference on*, pág. 216–225. IEEE, 2011.

[GGHS09] M. Guevara, C. Gregg, K. Hazelwood, e K. Skadron. Enabling task parallelism in the CUDA scheduler. In *Workshop on Programming Models for Emerging Architectures*, 2009.

[GKHM11] B. Gaster, D.R. Kaeli, L. Howes, e P. Mistry. *Heterogeneous Computing with OpenCL*. Morgan Kaufmann Pub, 2011.

[HLY+09] B. He, M. Lu, K. Yang, R. Fang, N.K. Govindaraju, Q. Luo, e P.V. Sander. Relational query coprocessing on graphics processors. *ACM Transactions on Database Systems (TODS)*, 34(4):21, 2009.

[Int11] Intel. Intel insights at SuperComputing11, 2011.

[KBI+09] M. Kulkarni, M. Burtscher, R. Inkulu, K. Pingali, e C. Casçaval. How much parallelism is there in irregular applications? In *ACM SIGPLAN Notices*, volume 44, pág. 3–14. ACM, 2009.

[KMSM12] Z. Koza, M. Matyka, S. Szkoda, e Ł. Mirosław. Compressed multiple-row storage format. *Arxiv preprint arXiv:1203.2946*, 2012.

[LH] A. Leist e KA Hawick. Graph generation on GPUs using dynamic memory allocation. In *Proc. International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'11)*.

[M+09] A. Munshi et al. The OpenCL specification. *Khronos OpenCL Working Group*, pág. l1–15, 2009.

[MBH+02] D.T. Marr, F. Binns, D.L. Hill, G. Hinton, D.A. Koufaty, J.A. Miller, e M. Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1):4–15, 2002.

[Mun08] A. Munshi. OpenCL: Parallel computing on the GPU and CPU. *SIGGRAPH, Tutorial*, 2008.

[MZZ+10] S. Mu, X. Zhang, N. Zhang, J. Lu, Y.S. Deng, e S. Zhang. IP routing processing with graphic processors. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pág. 93–98. European Design and Automation Association, 2010.

[NI09] A. Nottingham e B. Irwin. GPU packet classification using opencl: a consideration of viable classification methods. In *Proceedings of the 2009 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists*, pág. 160–169. ACM, 2009.

[NLKB11] S. Nobari, X. Lu, P. Karras, e S. Bressan. Fast random graph generation. In *Proceedings of the 14th International Conference on Extending Database Technology*, pág. 331–342. ACM, 2011.

[nVi08] C. nVidia. Compute unified device architecture–reference manual. *NVIDIA Corporation*, 2008.

[Saa94] Y. Saad. Sparskit: A basic toolkit for sparse matrix computations. *URL http://www-users. cs. umn. edu/saad*, 1994.

[WNDS99] Mason Woo, Jackie Neider, Tom Davis, e Dave Shreiner. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 1999.

[Wol10] M. Wolfe. Implementing the PGI accelerator model. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pág. 43–50. ACM, 2010.

[WZA+09] D. Wu, F. Zhang, N. Ao, F. Wang, X. Liu, e G. Wang. A batched GPU algorithm for set intersection. In *Pervasive Systems, Algorithms, and Networks (ISPAN), 2009 10th International Symposium on*, pág. 752–756. IEEE, 2009.

[ZFM] L. Zanotto, A. Ferreira, e M. Matsumoto. Arquitetura e programação de GPU nVidia.